Chapter 5

Parametric and Hypothetical Judgments

Many deductive systems employ reasoning from hypotheses. We have seen an example in Section 2.5: a typing derivation of a Mini-ML expression requires assumptions about the types of its free variables. Another example occurs in the system of natural deduction in Chapter ??, where a deduction of the judgment that $A \supset B$ *is true* can be given as a deduction of B *is true* from the hypothesis A *is true*. We refer to a judgment that J is derivable under a hypothesis J' as a *hypothetical judgment*. Its critical property is that we can substitute a derivation $\mathcal{D}$ of J' for every use of the hypothesis J' to obtain a derivation which no longer depends on the assumption J'.

Related is reasoning with parameters, which also occurs frequently. The system of natural deduction provides once again a typical example: we can infer that $\forall x.\ A$ *is true* if we can show that $[a/x]A$ *is true*, where a is a new parameter which does not occur in any undischarged hypothesis. Similarly, in the typing rules for Mini-ML we postulate that every variable is declared at most once in a context Γ, that is, in the rule

$$\frac{\Gamma, x{:}\tau_1 \triangleright e : \tau_2}{\Gamma \triangleright \mathbf{lam}\ x.\ e : \tau_1 \to \tau_2}\ \mathsf{tp\_lam}$$

the variable x is new with respect to Γ (which represents the hypotheses of the derivation). This side condition can always be fulfilled by tacitly renaming the bound variable. We refer to a judgment that J is derivable with parameter x as a *parametric judgment*. Its critical property is that we can substitute an expression t for x throughout a derivation of a parametric judgment to obtain a derivation which no longer depends on the parameter x.

Since parametric and hypothetical judgments are common, it is natural to ask if we can directly support them within the logical framework. The answer is affirmative—the key is the notion of function provided in LF. Briefly, the derivation of a hypothetical judgment is represented by a function which maps a derivation of the hypothesis to a derivation of the conclusion. Applying this function corresponds to substituting a derivation for appeals to the hypothesis. Similarly, the derivation of a parametric judgment is represented by a function which maps an expression to a derivation of the instantiated conclusion. Applying this function corresponds to substituting an expressions for the parameter throughout the parametric derivation.

In the remainder of this chapter we elaborate the notions of parametric and hypothetical judgment and their representation in LF. We also show how to exploit them to arrive at a natural and elegant representation of the proof of type preservation for Mini-ML.

5.1 Closed Expressions

When employing parametric and hypothetical judgments, we must formulate the representation theorems carefully in order to avoid paradoxes. As a simple example, we consider the judgment e *Closed* which expresses that e has no free variables. Expression constructors which do not introduce any bound variables are treated in a straightforward manner.

$$\frac{}{\mathbf{z}\ \textit{Closed}}\,\mathsf{clo\_z} \qquad\qquad \frac{e\ \textit{Closed}}{\mathbf{s}\ e\ \textit{Closed}}\,\mathsf{clo\_s}$$

$$\frac{e_1\ \textit{Closed} \qquad e_2\ \textit{Closed}}{\langle e_1, e_2\rangle\ \textit{Closed}}\,\mathsf{clo\_pair}$$

$$\frac{e\ \textit{Closed}}{\mathbf{fst}\ e\ \textit{Closed}}\,\mathsf{clo\_fst} \qquad\qquad \frac{e\ \textit{Closed}}{\mathbf{snd}\ e\ \textit{Closed}}\,\mathsf{clo\_snd}$$

$$\frac{e_1\ \textit{Closed} \qquad e_2\ \textit{Closed}}{e_1\ e_2\ \textit{Closed}}\,\mathsf{clo\_app}$$

In order to give a concise formulation of the judgment whenever variables are bound we use hypothetical judgments. For example, in order to conclude that **lam** $x.\ e$ is closed, we must show that e is closed under the assumption that x is closed. The hypothesis about x may only be used in the deduction of e, but not elsewhere. Furthermore, in order to avoid confusion between different bound variables with the same name, we would like to make sure that the name x is not already used, that is, the judgment should be parametric in x. The hypothetical judgment that J is derivable from hypotheses $J_1, \ldots, J_n$ is written as

$$\begin{array}{c} J_1 \;\ldots\; J_n \\ \vdots \\ J \end{array}$$

The construction of a deduction of a hypothetical judgment should be intuitively clear: in addition to the usual inference rules, we may also use a hypothesis as evidence for a judgment. But we must also indicate where an assumption is *discharged*, that is, after which point in a derivation it is no longer available. We indicate this by providing a name for the hypothesis J and labelling the inference at which the hypothesis is discharged correspondingly. Similarly, we label the inference at which a parameter is discharged. The remaining inference rules for the judgment e *Closed* using this notation are given below.

$$\dfrac{e_1\ \mathit{Closed} \qquad e_2\ \mathit{Closed} \qquad \begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u} \\ \vdots \\ e_3\ \mathit{Closed}\end{array}}{(\textbf{case}\ e_1\ \textbf{of}\ \textbf{z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3)\ \mathit{Closed}}\ \mathsf{clo\_case}^{x,u}$$

$$\dfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u} \\ \vdots \\ e\ \mathit{Closed}\end{array}}{\textbf{lam}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_lam}^{x,u} \qquad \dfrac{e_1\ \mathit{Closed} \qquad \begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u} \\ \vdots \\ e_2\ \mathit{Closed}\end{array}}{\textbf{let val}\ x = e_1\ \textbf{in}\ e_2\ \mathit{Closed}}\ \mathsf{clo\_letv}^{x,u}$$

$$\dfrac{e_1\ \mathit{Closed} \qquad \begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u} \\ \vdots \\ e_2\ \mathit{Closed}\end{array}}{\textbf{let name}\ x = e_1\ \textbf{in}\ e_2\ \mathit{Closed}}\ \mathsf{clo\_letn}^{x,u} \qquad \dfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u} \\ \vdots \\ e\ \mathit{Closed}\end{array}}{\textbf{fix}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_fix}^{x,u}$$

In order to avoid ambiguity we assume that in a given deduction, all labels for the inference rules clo_case, clo_lam, clo_letv, clo_letn and clo_fix are distinct. An alternative to this rather stringent, but convenient requirement is suggestive of the representation of hypothetical judgments in LF: we can think of a label u as a variable ranging over deductions. The variable is bound by the inference which discharges the hypothesis.

The following derivation shows that the expression $\mathbf{let\ name}\ f = \mathbf{lam}\ x.\ x\ \mathbf{in}\ f\ (f\ \mathbf{z})$ is closed.

$$\dfrac{\dfrac{\dfrac{}{x\ \mathit{Closed}}\,u}{\mathbf{lam}\ x.\ x\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u} \qquad \dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \quad \dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \quad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}}{\mathbf{let\ name}\ f = \mathbf{lam}\ x.\ x\ \mathbf{in}\ f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_letn}^{f,w}$$

This deduction has no undischarged assumptions, but it contains subderivations with hypotheses. The right subderivation, for example, would traditionally be written as

$$\dfrac{f\ \mathit{Closed} \qquad \dfrac{f\ \mathit{Closed} \quad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}.$$

In this notation we can not determine if there are two hypotheses (which happen to coincide) or two uses of the same hypothesis. This distinction may be irrelevant under some circumstances, but in many situations it is critical. Therefore we retain the labels even for hypothetical derivations, with the restriction that the free labels must be used consistently, that is, all occurrences of a label must justify the same hypothesis. The subderivation above then reads

$$\dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \quad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}.$$

There are certain reasoning principles for hypothetical derivations which are usually not stated explicitly. One of them is that hypotheses need not be used. For example, $\mathbf{lam}\ x.\ \mathbf{z}$ is closed as witnessed by the derivation

$$\dfrac{\dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{\mathbf{lam}\ x.\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}$$

which contains a subdeduction of $\mathbf{z}$ *Closed* from hypothesis $u :: x$ *Closed*. Another principle is that hypotheses may be used more than once and thus, in fact, arbitrarily often. Finally, the order of the hypotheses is irrelevant (although their labelling is not). This means that a hypothetical deduction in this notation could be evidence for a variety of hypothetical judgments which differ in the order of the hypotheses or may contain further, unused hypotheses. One can make these principles explicit as inference rules, in which case we refer to them as *weakening* (hypotheses need not be used), *contraction* (hypotheses may be used more than once), and *exchange* (the order of the hypotheses is irrelevant). We should keep in mind that if these principles do not apply then the judgment should not be considered to be hypothetical in the usual sense, and the techniques below may not apply. These properties have been studied abstractly as *consequence relations* [Gar92].

The example derivation above is not only hypothetical in w, but also parametric in f, and we can therefore substitute an expression such as $\mathbf{lam}\,x.\,x$ for f and obtain another valid deduction.

$$\dfrac{\dfrac{}{\mathbf{lam}\,x.\,x\ \mathit{Closed}}\,w \qquad \dfrac{\dfrac{}{\mathbf{lam}\,x.\,x\ \mathit{Closed}}\,w \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{(\mathbf{lam}\,x.\,x)\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{(\mathbf{lam}\,x.\,x)\ ((\mathbf{lam}\,x.\,x)\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}$$

If $\mathcal{C} :: e\ \mathit{Closed}$ is parametric in x, then we write $[e'/x]\mathcal{C} :: [e'/x]e\ \mathit{Closed}$ for the result of substituting e' for x in the deduction $\mathcal{C}$. In the example, the deduction still depends on the hypothesis w, which suggests another approach to understanding hypothetical judgments. If a deduction depends on a hypothesis $u :: J$ we can substitute any valid deduction of J for this hypothesis to obtain another deduction which no longer depends on $u :: J$. Let $\mathcal{C}$ be the deduction above and let $\mathcal{C}' :: \mathbf{lam}\,x.\,x\ \mathit{Closed}$ be

$$\dfrac{\dfrac{}{x\ \mathit{Closed}}\,u}{\mathbf{lam}\,x.\,x\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}.$$

Note that this deduction is *not* parametric in x, that is, x must be considered a bound variable within $\mathcal{C}'$. The result of substituting $\mathcal{C}'$ for w in $\mathcal{C}$ is

$$\dfrac{\dfrac{\dfrac{}{x\ \mathit{Closed}}\,u}{\mathbf{lam}\,x.\,x\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u} \qquad \dfrac{\dfrac{\dfrac{}{x'\ \mathit{Closed}}\,u'}{\mathbf{lam}\,x'.\,x'\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x',u'} \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{(\mathbf{lam}\,x'.\,x')\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{(\mathbf{lam}\,x.\,x)\ ((\mathbf{lam}\,x'.\,x')\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}$$

where we have renamed some occurrences of x and u in order to satisfy our global side conditions on parameter names. In general we write $[\mathcal{D}'/u]\mathcal{D}$ for the result of substituting $\mathcal{D}'$ for the hypothesis $u :: J'$ in $\mathcal{D}$, where $\mathcal{D}' :: J'$. During substitution we may need to rename parameters or labels of assumptions to avoid violating side conditions on inference rules. This is analogous to the renaming of bound variables during substitution in terms in order to avoid variable capture.

The representation of the judgment e *Closed* in LF follows the judgment-as-types principle: we introduce a type family 'closed' indexed by an expression.

$$\mathsf{closed} \;:\; \mathsf{exp} \to \mathsf{type}$$

The inference rules that do not employ hypothetical judgments are represented straightforwardly.

$$\begin{array}{lll}
\mathsf{clo_z} & : & \mathsf{closed\ z} \\
\mathsf{clo_s} & : & \Pi E{:}\mathsf{exp}\mathsf{closed}\ E \to \mathsf{closed}\ (\mathsf{s}\ E) \\
\mathsf{clo_pair} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}. \\
 & & \quad \mathsf{closed}\ E_1 \to \mathsf{closed}\ E_2 \to \mathsf{closed}\ (\mathsf{pair}\ E_1\ E_2) \\
\mathsf{clo_fst} & : & \Pi E{:}\mathsf{exp}.\ \mathsf{closed}\ E \to \mathsf{closed}\ (\mathsf{fst}\ E) \\
\mathsf{clo_snd} & : & \Pi E{:}\mathsf{exp}.\ \mathsf{closed}\ E \to \mathsf{closed}\ (\mathsf{snd}\ E) \\
\mathsf{clo_app} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}. \\
 & & \quad \mathsf{closed}\ E_1 \to \mathsf{closed}\ E_2 \to \mathsf{closed}\ (\mathsf{app}\ E_1\ E_2)
\end{array}$$

Now we reconsider the rule $\mathsf{clo\_lam}$.

$$\dfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}\ u \\ \vdots \\ e\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_lam}^{x,u}$$

The judgment in the premiss is parametric in x and hypothetical in x *Closed*. We thus consider it as a function which, when applied to an e' and a deduction $\mathcal{C} :: e'$ *Closed* yields a deduction of $[e'/x]e$ *Closed*.

$$\begin{array}{lll}
\mathsf{clo_lam} & : & \Pi E{:}\mathsf{exp} \to \mathsf{exp}. \\
 & & \quad (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E\ x)) \to \mathsf{closed}\ (\mathsf{lam}\ E)
\end{array}$$

Recall that it is necessary to represent the scope of a binding operator in the language of expressions as a function from expressions to expressions. Similar declarations are necessary for the hypothetical judgments in the premisses of the clo_letv, clo_letn, and clo_fix rules.

$$
\begin{array}{lll}
\mathsf{clo\_case} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \Pi E_3{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad \mathsf{closed}\ E_1 \to \mathsf{closed}\ E_2 \\
& & \quad \to (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E_3\ x)) \\
& & \quad \to \mathsf{closed}\ (\mathsf{case}\ E_1\ E_2\ E_3) \\
\mathsf{clo\_letv} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad \mathsf{closed}\ E_1 \to (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E_2\ x)) \\
& & \quad \to \mathsf{closed}\ (\mathsf{letv}\ E_1\ E_2) \\
\mathsf{clo\_letn} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad \mathsf{closed}\ E_1 \to (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E_2\ x)) \\
& & \quad \to \mathsf{closed}\ (\mathsf{letn}\ E_1\ E_2) \\
\mathsf{clo\_fix} & : & \Pi E{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E\ x)) \to \mathsf{closed}\ (\mathsf{fix}\ E)
\end{array}
$$

We refer to the signature which includes expression constructors, the declarations of the family closed and the encodings of the inference rules above as EC.

In order to appreciate how this representation works, it is necessary to understand the representation function $\ulcorner\cdot\urcorner$ on deductions. As usual, the definition of the representation function follows the structure of $\mathcal{C} :: e\ \textit{Closed}$. We only show a few typical cases.

Case: $\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1 \\ e_1\ \textit{Closed}\end{array} \qquad \begin{array}{c}\mathcal{C}_2 \\ e_2\ \textit{Closed}\end{array}}{e_1\ e_2\ \textit{Closed}}\,\mathsf{clo\_app}$. Then

$$\ulcorner\mathcal{C}\urcorner = \mathsf{clo\_app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{C}_1\urcorner\ \ulcorner\mathcal{C}_2\urcorner.$$

Case: $\mathcal{C} = \dfrac{\begin{array}{c}\overline{x\ \textit{Closed}}\,u \\ \mathcal{C}_1 \\ e\ \textit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \textit{Closed}}\,\mathsf{clo\_lam}^{x,u}$. Then

$$\ulcorner\mathcal{C}\urcorner = \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}_1\urcorner).$$

Case: $\mathcal{C} = \dfrac{}{x\ \textit{Closed}}\,u$. Then

$$\ulcorner\mathcal{C}\urcorner = u.$$

The example deduction above which is evidence for the judgment **let name** $f =$ **lam** $x.\ x$ **in** f **z** *Closed* is represented as

$$
\begin{array}{ll}
\vdash_{EC} & \mathsf{clo\_letn}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ x))\ (\lambda f{:}\mathsf{exp}.\ \mathsf{app}\ f\ \mathsf{z}) \\
& \quad (\mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ x)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ u.\ u)) \\
& \quad (\lambda f{:}\mathsf{exp}.\ \lambda w{:}\mathsf{closed}\ f.\ \mathsf{clo\_app}\ f\ \mathsf{z}\ w\ \mathsf{clo\_z}) \\
& : \mathsf{closed}\ (\mathsf{letn}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ x))\ (\lambda f{:}\mathsf{exp}.\ \mathsf{app}\ f\ \mathsf{z}))
\end{array}
$$

To show that the above is derivable we need to employ the rule of type conversion as in the example on page 59. The naive formulation of the soundness of the representation does not take the hypothetical or parametric judgments into account.

Property 5.1 (Soundness of Representation, Version 1)
Given any deduction $\mathcal{C} :: e$ Closed. Then $\vdash_{EC} \ulcorner\mathcal{C}\urcorner \Uparrow \mathsf{closed}\ \ulcorner e\urcorner$.

While this indeed a theorem, it cannot be proven directly by induction—the induction hypothesis will not be strong enough to deal with the inference rules whose premisses require deductions of hypothetical judgments. In order to formulate and prove a more general property we have to consider the representation of hypothetical judgments. As one can see from the example above, the deduction fragment

$$\dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}$$

is represented by the LF object

$$\mathsf{clo\_app}\ f\ \mathsf{z}\ w\ \mathsf{clo\_z}$$

which is valid in the context with the declarations $f{:}\mathsf{exp}$ and $w{:}\mathsf{closed}\ f$. In order to make the connection between the hypotheses and the LF context explicit, we retain the labels of the assumptions and explicitly define the representation of a list of hypotheses. Let $\Delta = u_1 :: x_1\ \mathit{Closed}, \ldots, u_n :: x_n\ \mathit{Closed}$ be a list of hypotheses where all labels are distinct. Then

$$\ulcorner\Delta\urcorner = x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{exp}, u_n{:}\mathsf{closed}\ x_n.$$

The reformulated soundness property now references the available hypotheses.

Property 5.2 (Soudness of Representation, Version 2)
Given any deduction $\mathcal{C} :: e$ Closed from hypotheses $\Delta = u_1 :: x_1$ Closed, $\ldots, u_n :: x_n$ Closed. Then $\vdash_{EC} \ulcorner\Delta\urcorner$ Ctx and

$$\ulcorner\Delta\urcorner \vdash_{EC} \ulcorner\mathcal{C}\urcorner \Uparrow \mathsf{closed}\ \ulcorner e\urcorner.$$

Proof: The proof is by induction on the structure of $\mathcal{C}$. We show three typical cases.
Case:

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1\\ e_1\ \mathit{Closed}\end{array} \qquad \begin{array}{c}\mathcal{C}_2\\ e_2\ \mathit{Closed}\end{array}}{e_1\ e_2\ \mathit{Closed}}\,\mathsf{clo\_app}.$$

Since $\mathcal{C}$ is a deduction from hypotheses Δ, both $\mathcal{C}_1$ and $\mathcal{C}_2$ are also deductions from hypotheses Δ. From the induction hypothesis we conclude then that

1. $\ulcorner\Delta\urcorner \vdash_{EC} \ulcorner\mathcal{C}_1\urcorner \Uparrow \mathsf{closed}\ \ulcorner e_1\urcorner$, and
2. $\ulcorner\Delta\urcorner \vdash_{EC} \ulcorner\mathcal{C}_2\urcorner \Uparrow \mathsf{closed}\ \ulcorner e_2\urcorner$

are both derivable. Thus, from the type of clo_app,

$$\ulcorner\Delta\urcorner \vdash_{EC} \mathsf{clo\_app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{C}_1\urcorner\ \ulcorner\mathcal{C}_2\urcorner \Uparrow \mathsf{closed}\ (\mathsf{app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner).$$

It remains to notice that $\ulcorner e_1\ e_2\urcorner = \mathsf{app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner$.

Case:

$$\mathcal{C} = \cfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}\,u\\ \mathcal{C}_1\\ e\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}.$$

Then $\mathcal{C}_1$ is a deduction from hypotheses $\Delta, u :: x\ \mathit{Closed}$, and

$$\ulcorner\Delta, u :: x\ \mathit{Closed}\urcorner = \ulcorner\Delta\urcorner, x{:}\mathsf{exp}, u{:}\mathsf{closed}\ x.$$

By the induction hypothesis on $\mathcal{C}_1$ we thus conclude that

$$\ulcorner\Delta\urcorner, x{:}\mathsf{exp}, u{:}\mathsf{closed}\ x \vdash_{EC} \ulcorner\mathcal{C}_1\urcorner \Uparrow \mathsf{closed}\ \ulcorner e\urcorner$$

is derivable. Hence, by two applications of the canpi rule for canonical forms,

$$\ulcorner\Delta\urcorner \vdash_{EC} \lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}_1\urcorner \Uparrow \Pi x{:}\mathsf{exp}.\ \Pi u{:}\mathsf{closed}\ x.\ \mathsf{closed}\ \ulcorner e\urcorner$$

is also derivable.

By the representation theorem for expressions (Theorem 3.6) and the weakening for LF we also know that

$$\ulcorner\Delta\urcorner \vdash_{EC} \lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner \Uparrow \mathsf{exp} \to \mathsf{exp}$$

is derivable. From this and the type of clo_lam we infer

$$\begin{array}{lll}\ulcorner\Delta\urcorner & \vdash_{EC} & \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\\ & & \quad\downarrow (\Pi x{:}\mathsf{exp}.\ \Pi u{:}\mathsf{closed}\ x.\ \mathsf{closed}\ ((\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ x))\\ & & \qquad \to \mathsf{closed}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)).\end{array}$$

By the rule atmcnv, using one β-conversion in the type above, we conclude

$$\begin{array}{lll}\ulcorner\Delta\urcorner & \vdash_{EC} & \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\\ & & \quad\downarrow (\Pi x{:}\mathsf{exp}.\ \Pi u{:}\mathsf{closed}\ x.\ \mathsf{closed}\ \ulcorner e\urcorner)\\ & & \qquad \to \mathsf{closed}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)).\end{array}$$

Using the rules atmapp and cancon which are now applicable we infer

$$\begin{array}{ll} \ulcorner\Delta\urcorner \quad \vdash_{EC} & \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}_1\urcorner) \\ & \Uparrow \mathsf{closed}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)), \end{array}$$

which is the desired conclusion since $\ulcorner \mathbf{lam}\ x.\ e\urcorner = \mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)$.

Case:

$$\mathcal{C} = \frac{}{x\ \mathit{Closed}}\,u.$$

Then $\ulcorner\mathcal{C}\urcorner = u$, to which $\ulcorner\Delta\urcorner$ assigns type $\mathsf{closed}\ x = \mathsf{closed}\ \ulcorner x\urcorner$, which is what we needed to show.

□

The inverse of the representation function, $\llcorner\cdot\lrcorner$, is defined on canonical objects C of type $\mathsf{closed}\ E$ for some E of type exp. This is sufficient for the adequacy theorem below—one can extend it to arbitrary valid objects via conversion to canonical form. Again, we only show three critical cases.

$$\llcorner \mathsf{clo\_app}\ E_1\ E_2\ C_1\ C_2 \lrcorner \;=\; \frac{\llcorner C_1\lrcorner \qquad \llcorner C_2\lrcorner}{\llcorner E_1\lrcorner\ \llcorner E_2\lrcorner\ \mathit{Closed}}\,\mathsf{clo\_app}$$

$$\llcorner \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ E)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ C_1)\lrcorner \;=\; \frac{\begin{array}{c}\dfrac{}{x\ \mathit{Closed}}\,u \\ \llcorner C_1\lrcorner \\ \llcorner E\lrcorner\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ \llcorner E\lrcorner\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}$$

$$\llcorner u\lrcorner \;=\; \frac{}{x\ \mathit{Closed}}\,u$$

The last case reveals that the inverse of the representation function should be parameterized by a context so we can find the x which is assumed to be closed according to hypothesis u. Alternatively, we can assume that we always know the type of the canonical object we are translating to a deduction. Again, we are faced with the problem that the natural theorem regarding the function $\llcorner\cdot\lrcorner$ cannot be proved directly by induction.

Property 5.3 (Completeness of Representation, Version 1) *Given LF objects E and C such that $\vdash_{EC} E \Uparrow \mathsf{exp}$ and $\vdash_{EC} C \Uparrow \mathsf{closed}\ E$. Then $\llcorner C\lrcorner :: \llcorner E\lrcorner$ Closed.*

In order to prove this, we generalize it to allow appropriate contexts. These contexts need to be translated to an appropriate list of hypotheses. Let Γ be a context of the form $x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{closed}\ x_n$. Then $\llcorner\Gamma\lrcorner$ is the list of hypotheses $u_1 :: x_1\ \mathit{Closed}, \ldots, u_n :: x_n\, \mathit{Closed}$.

Property 5.4 (Completeness of Representation, Version 2) *Given a context* $\Gamma = x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{closed}\ x_n$ *and LF objects* E *and* C *such that* $\Gamma \vdash_{EC} E \Uparrow \mathsf{exp}$ *and* $\Gamma \vdash_{EC} C \Uparrow \mathsf{closed}\ E$. *Then* $\llcorner C \lrcorner :: \llcorner E \lrcorner$ *Closed is a valid deduction from hypotheses* $\llcorner\Gamma\lrcorner$. *Moreoever,* $\llcorner\ulcorner\mathcal{C}\urcorner\lrcorner = \mathcal{C}$ *and* $\llcorner\ulcorner\Delta\urcorner\lrcorner = \Delta$ *for deductions* $\mathcal{C} :: e$ *Closed and hypotheses* Δ.

Proof: By induction on the structure of the derivation $\Gamma \vdash_{EC} C \Uparrow \mathsf{closed}\ E$. The restriction to contexts Γ of a certain form is crucial in this proof (see Exercise 5.1). □

The usual requirement that $\ulcorner\cdot\urcorner$ be a compositional bijection can be understood in terms of substitution for deductions. Let $\mathcal{C} :: e$ *Closed* be a deduction from hypothesis $u :: x$ *closed*. Then compositionality of the representation function requires

$$\ulcorner[\mathcal{C}'/u][e'/x]\mathcal{C}\urcorner = [\ulcorner\mathcal{C}'\urcorner/u][\ulcorner e'\urcorner/x]\ulcorner\mathcal{C}\urcorner$$

whenever $\mathcal{C}' :: e'$ *Closed*. Note that the substitution on the left-hand side is substitution for undischarged hypotheses in a deduction, while substitution on the right is at the level of LF objects. Deductions of parametric and hypothetical judgments are represented as functions in LF. Applying such functions means to substitute for deductions, which can be exhibited if we rewrite the right-hand side of the equation above, preserving definitional equality.

$$[\ulcorner\mathcal{C}'\urcorner/u][\ulcorner e'\urcorner/x]\ulcorner\mathcal{C}\urcorner \equiv (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}\urcorner)\ \ulcorner e'\urcorner\ \ulcorner\mathcal{C}'\urcorner$$

The discipline of dependent function types ensures the validity of the object on the right-hand side:

$$(\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}\urcorner)\ \ulcorner e'\urcorner : [\ulcorner e'\urcorner/x](\mathsf{closed}\ x \to \mathsf{closed}\ \ulcorner e\urcorner).$$

Hence, by compositionality of the representation for expressions,

$$(\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}\urcorner)\ \ulcorner e'\urcorner : \mathsf{closed}\ \ulcorner e'\urcorner \to \mathsf{closed}\ \ulcorner[e'/x]e\urcorner$$

and the application of this object to $\ulcorner\mathcal{C}'\urcorner$ is valid and of the appropriate type.

Theorem 5.5 (Adequacy) *There is a bijection between deductions* $\mathcal{C} :: e$ *Closed from hypotheses* $u_1 :: x_1$ *Closed*, $\ldots, u_n :: x_n$ *Closed and LF objects* C *such that*

$$x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{exp}, u_n{:}\mathsf{closed}\ x_n \vdash_{EC} C \Uparrow \mathsf{closed}\ \ulcorner e\urcorner.$$

The bijection is compositional in the sense that for an expression e_i and a deduction $\mathcal{C}_i :: e_i$ Closed, we have

$$\ulcorner [\mathcal{C}_i/u_i][e_i/x_i]\mathcal{C} \urcorner = [\ulcorner \mathcal{C}_i \urcorner/u_i][\ulcorner e_i \urcorner/x_i]\ulcorner \mathcal{C} \urcorner$$

Proof: Properties 5.2 and 5.4 show the existence of the bijection. To show that it is compositional we reason by induction over the structure of $\mathcal{C}$ (see Exercise 5.2). □

5.2 Function Types as Goals in Elf

Below we give the transcription of the LF signature above in Elf.

```
closed : exp -> type.  %name closed U u.

% Natural Numbers
clo_z     : closed z.
clo_s     : closed (s E)
             <- closed E.
clo_case  : closed (case E1 E2 E3)
             <- closed E1
             <- closed E2
             <- ({x:exp} closed x -> closed (E3 x)).

% Pairs
clo_pair : closed (pair E1 E2)
              <- closed E1
              <- closed E2.
clo_fst  : closed (fst E)
              <- closed E.
clo_snd  : closed (snd E)
              <- closed E.

% Functions
clo_lam : closed (lam E)
             <- ({x:exp} closed x -> closed (E x)).
clo_app : closed (app E1 E2)
             <- closed E1
             <- closed E2.


% Definitions
```

```
clo_letv : closed (letv E1 E2)
             <- closed E1
             <- ({x:exp} closed x -> closed (E2 x)).
clo_letn : closed (letn E1 E2)
             <- closed E1
             <- ({x:exp} closed x -> closed (E2 x)).

% Recursion
clo_fix : closed (fix E)
            <- ({x:exp} closed x -> closed (E x)).
```

Note that we have changed the order of arguments as in other examples. It seems reasonable to expect that this signature could be used as a program to determine if a given object e of type `exp` is closed. Let us consider the subgoals as they arise in a query to check if **lam** $y.\, y$ is closed.

```
?- closed (lam [y:exp] y).
% Resolved with clause clo_lam
?- {x:exp} closed x -> closed (([y:exp] y) x).
```

Recall that solving a goal means to find a closed expression of the query type. Here, the query type is a (dependent) function type. From Theorem 3.13 we know that if a closed object of type $\Pi x{:}A.\ B$ exists, then there is a definitionally equal object of the form $\lambda x{:}A.\ M$ such that M has type B in the context augmented with the assumption $x{:}A$. It is thus a complete strategy in this case to make the assumption that `x` has type `exp` and solve the goal

```
?- closed x -> closed (([y:exp] y) x).
```

However, `x` now is not a free variable in same sense as `V` in the query

```
?- eval (lam [y:exp] y) V.
```

since it is not subject to instantiation during unification. In order to distinguish these different kinds of variables, we call variables which are subject to instantiation *logic variables* and variables which act as constants to unification *parameters*. Unlike logic variables, parameters are shown as lower-case constants. Thus the current goal might be presented as

```
x : exp
?- closed x -> closed (([y:exp] y) x).
```

Here we precede the query with the typings for the current parameters. Now recall that `A -> B` is just a concrete syntax for `{_:A} B` where `_` is an anonymous variable which cannot appear free in `B`. Thus, this case is handled similarly: we introduce a new parameter `u` of type `closed x` and then solve the subgoal

```
x : exp
u : closed x
?- closed (([y:exp] y) x).
```

By an application of β-conversion this is transformed into the equivalent goal

```
x : exp
u : closed x
?- closed x.
```

Now we can use the parameter `u` as the requested object of type `closed x` and the query succeeds without further subgoals.

We now briefly consider, how the appropriate closed object of the original query, namely `?- closed (lam [y:exp] y).` would be constructed. Recall that if $\Gamma, x{:}A \vdash_{\Sigma} M : B$ then $\Gamma \vdash_{\Sigma} \lambda x{:}A.\ M : \Pi x{:}A.\ B$. Using this we can now through the trace of the search in reverse, constructing inhabiting objects as we go along and inserting conversions where necessary.

```
                  u : closed x.
      [u:closed x] u : closed x -> closed x.
[x:exp][u:closed x] u : {x:exp} closed x -> closed x.
[x:exp][u:closed x] u : {x:exp} closed x -> closed (([y:exp] y) x).
clo_lam ([x:exp][u:closed x] u)
                     : closed (lam [y:exp] y).
```

Just as in Prolog, search proceeds according to a fixed operational semantics. This semantics specifies that clauses (that is, LF constant declarations) are tried in order from the first to the last. Before referring to the fixed signature, however, the temporary hypotheses are consulted, always considering the most recently introduced parameter first. After all of them have been considered, then the current signature is traversed. In this example the search order happens to be irrelevant as there will always be at most one assumption available for any expression parameter.

The representations of parametric and hypothetical judgments can also be given directly at the top-level. Here are two examples: the first to find the representation of the hypothetical deduction of $f\ (f\ \mathbf{z})$ *closed* from the hypothesis f *closed*, the second to illustrate failure when given an expression ($\langle x, x\rangle$) which is not closed.

```
?- Q : {f:exp} closed f -> closed (app f (app f z)).

Q = [f:exp] [u:closed f] clo_app (clo_app clo_z u) u.
More? y
no more solutions
?- Q : {x:exp} closed (pair x x).

no
```

Note that the quantification on the variable `x` is necessary, since the query `?- closed (pair x x).` is considered to contain an undeclared constant `x` (which is an error), and the query `?- closed (pair X X)` considers `X` as a logic variable subject to instantiation:

```
?- Q : closed (pair X X).
Solving...

X = z,
Q = clo_pair clo_z clo_z.
More? y

X = s z,
Q = clo_pair (clo_s clo_z) (clo_s clo_z).

yes
```

5.3 Negation

Now that we have seen how to write a program to detect closed expressions, how do we write a program which succeeds if an expression is *not* closed? In Prolog, one has the possibility of using the unsound technique of negation-as-failure to write a predicate which succeeds if and only if another predicate fails finitely. In Elf, this technique is not available. Philosophically one might argue that the absence of evidence for e *Closed* does not necessarily mean that e is not closed. More pragmatically, note that if we possess evidence that e is closed, then this will continue to be evidence regardless of any further inference rules or hypotheses we might introduce to demonstrate that expressions are closed. However, the judgment that $\langle x, x\rangle$ is *not* closed does not persist if we add the hypothesis that x is closed. Only under a so-called *closed-world assumption*, that is, the assumption that no further hypotheses or inference rules will be considered, is it reasonable to conclude the $\langle x, x\rangle$ is not closed. The philosophy behind the logical framework is that we work with an implicit *open-world assumption*, that is, all judgments, once judged to be evident since witnessed by a deduction, should remain evident under extensions of the current rules of inference. Note that this is clearly *not* the case for the meta-theorems we prove. Their proofs rely on induction on the structure of derivations and they may no longer be valid when further rules are added.

Thus it is necessary to explicitly define a judgment e *Open* to provide means for giving evidence that e is open, that is, it contains at least one free variable. Below is the implementation of such a judgment in Elf.

```
open : exp -> type.  %name open v
```

```
% Natural Numbers
open_s     : open (s E) <- open E.
open_case1 : open (case E1 E2 E3) <- open E1.
open_case2 : open (case E1 E2 E3) <- open E2.
open_case3 : open (case E1 E2 E3) <- ({x:exp} open (E3 x)).

% Pairs
open_pair1 : open (pair E1 E2) <- open E1.
open_pair2 : open (pair E1 E2) <- open E2.
open_fst   : open (fst E) <- open E.
open_snd   : open (snd E) <- open E.

% Functions
open_lam  : open (lam E) <- ({x:exp} open (E x)).
open_app1 : open (app E1 E2) <- open E1.
open_app2 : open (app E1 E2) <- open E2.

% Definitions
open_letv1 : open (letv E1 E2) <- open E1.
open_letv2 : open (letv E1 E2) <- ({x:exp} open (E2 x)).
open_letn1 : open (letn E1 E2) <- open E1.
open_letn2 : open (letn E1 E2) <- ({x:exp} open (E2 x)).

% Recursion
open_fix : open (fix E) <- ({x:exp} open (E x)).
```

One curious fact about this judgment is that there is no base case, that is, without any hypotheses any query of the form `?- open` $\ulcorner e \urcorner$. will fail! That is, with a given query we must provide evidence that any parameters which may occur in it are open. For example,

```
?- Q : {x:exp} open (pair x x).
no
?- Q : {x:exp} open x -> open (pair x x).

Empty Substitution.
Q = [x:exp] [v:open x] open_pair1 v.
More? y
Empty Substitution.
Q = [x:exp] [v:open x] open_pair2 v.
More? y
No more solutions
```

```
?- Q : {x:exp} open x -> open (lam [x:exp] pair x x).
no
```

5.4 Representing Mini-ML Typing Derivations

In this section we will show a natural representation of Mini-ML typing derivations in LF. In order to avoid confusion between the contexts of LF and the contexts of Mini-ML, we will use Δ throughout the remainder of this chapter to designate a Mini-ML context. The typing judgment of Mini-ML then has the form

$$\Delta \rhd e : \tau$$

and expresses that e has type τ in context Δ. We observe that this judgment can be interpreted as a hypothetical judgment with hypotheses Δ. There is thus an alternative way to describe the judgment $e : \tau$ which employs hypothetical judgments without making assumptions explicit.

$$\frac{\Delta, x{:}\tau_1 \rhd e : \tau_2}{\Delta \rhd \mathbf{lam}\, x.\, e : \tau_1 \to \tau_2}\,\mathsf{tp\_lam} \qquad \frac{\begin{array}{c}\overline{\rhd x : \tau_1}\,u \\ \vdots \\ \rhd e : \tau_2\end{array}}{\rhd \mathbf{lam}\, x.\, e : \tau_1 \to \tau_2}\,\mathsf{tp\_lam}^{x,u}$$

The judgment in the premiss in the formulation of the rule on the right is parametric in x and hypothetical in $u :: x{:}\tau_1$. On the left, all available hypothesis are represented explicitly. The restriction that each variable may be declared at most once in a context and bound variables may be renamed tacitly encodes the parametricity with respect to x.

First, however, the representation of Mini-ML types. We declare an LF type constant, tp, for the representation of Mini-ML types. Recall, from Section 2.5,

$$\text{Types} \quad \tau \quad ::= \quad \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha$$

It is important to bear in mind that $\to$ is overloaded here, since it stands for the function type constructor in Mini-ML and in LF. It should be clear from the context which constructor is meant in each instance. The representation function and the LF declarations are straightforward.

$$\begin{array}{rcl}
 & & \\
\ulcorner \alpha \urcorner & = & \alpha \\
\ulcorner \mathbf{nat} \urcorner & = & \mathsf{nat} \\
\ulcorner \tau_1 \times \tau_2 \urcorner & = & \mathsf{cross}\, \ulcorner \tau_1 \urcorner\, \ulcorner \tau_2 \urcorner \\
\ulcorner \tau_1 \to \tau_2 \urcorner & = & \mathsf{arrow}\, \ulcorner \tau_1 \urcorner\, \ulcorner \tau_2 \urcorner
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{tp} & : & \mathsf{type} \\
 & & \\
\mathsf{nat} & : & \mathsf{tp} \\
\mathsf{cross} & : & \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp} \\
\mathsf{arrow} & : & \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}
\end{array}$$

Here α on the right-hand side stands for a variable named α in the LF type theory. We refer to the signature in the right-hand column as T. We briefly state (without proof) the representation theorem.

Theorem 5.6 (Adequacy) *The representation function* $\ulcorner\cdot\urcorner$ *is a compositional bijection between Mini-ML types and canonical LF objects of type* tp *over the signature* T.

Now we try to apply the techniques for representing hypothetical judgments developed in Section 5.1 to the representation of the typing judgment (for an alternative, see Exercise 5.6). The representation will be as a type family 'of' such that

$$\ulcorner\Delta\urcorner \vdash \ulcorner\mathcal{P}\urcorner : \mathsf{of}\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner$$

whenever $\mathcal{P}$ is a deduction of $\Delta \triangleright e : \tau$. Thus,

$$\mathsf{of} \quad : \quad \mathsf{exp} \to \mathsf{tp} \to \mathsf{type}$$

with the representation for Mini-ML contexts Δ as LF contexts $\ulcorner\Delta\urcorner$.

$$\begin{array}{rcl} \ulcorner\cdot\urcorner & = & \cdot \\ \ulcorner\Delta, x{:}\tau\urcorner & = & \ulcorner\Delta\urcorner, x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ \ulcorner\tau\urcorner \end{array}$$

Here u must be chosen to be different from x and any other variable in $\ulcorner\Delta\urcorner$ in order to satisfy the general assumption about LF contexts. This assumption can be satisfied since we made a similar assumption about Δ.

For typing derivation themselves, we only show three critical cases in the definition of $\ulcorner\mathcal{P}\urcorner$ for $\mathcal{P} :: \Delta \triangleright e : \tau$. The remainder is given directly in Elf later. The type family $\mathsf{of} : \mathsf{exp} \to \mathsf{tp} \to \mathsf{type}$ represents the judgment $e : \tau$.

Case:

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1 \\ \Delta \triangleright e_1 : \tau_2 \to \tau_1\end{array} \qquad \begin{array}{c}\mathcal{P}_2 \\ \Delta \triangleright e_2 : \tau_2\end{array}}{\Delta \triangleright e_1\ e_2 : \tau_1}\ \mathsf{tp\_app}.$$

In this simple case we let

$$\ulcorner\mathcal{P}\urcorner = \mathsf{tp\_app}\ \ulcorner\tau_1\urcorner\ \ulcorner\tau_2\urcorner\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{P}_1\urcorner\ \ulcorner\mathcal{P}_2\urcorner$$

where

$$\begin{array}{lcl} \mathsf{tp\_app} & : & \Pi T_1{:}\mathsf{tp}.\ \Pi T_2{:}\mathsf{tp}.\ \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp} \\ & & \quad \mathsf{of}\ E_1\ (\mathsf{arrow}\ T_2\ T_1) \to \mathsf{of}\ E_2\ T_2 \to \mathsf{of}\ (\mathsf{app}\ E_1\ E_2)\ T_1 \end{array}$$

Case:

$$\mathcal{P} = \cfrac{\begin{array}{c}\mathcal{P}'\\ \Delta, x{:}\tau_1 \triangleright e : \tau_2\end{array}}{\Delta \triangleright \textbf{lam}\, x.\, e : \tau_1 \to \tau_2}\,\mathsf{tp\_lam}.$$

In this case we view $\mathcal{P}'$ as a deduction of a hypothetical judgment, that is, a derivation of $e : \tau_2$ from the hypothesis $x{:}\tau_1$. We furthermore note that $\mathcal{P}'$ is parametric in x and choose an appropriate functional representation.

$$\ulcorner\mathcal{P}\urcorner \quad = \quad \mathsf{tp\_lam}\, \ulcorner\tau_1\urcorner\, \ulcorner\tau_2\urcorner\, (\lambda x{:}\mathsf{exp}.\, \ulcorner e\urcorner)\, (\lambda x{:}\mathsf{exp}.\, \lambda u{:}\mathsf{of}\; x\, \ulcorner\tau_1\urcorner.\, \ulcorner\mathcal{P}'\urcorner)$$

The constant $\mathsf{tp\_lam}$ must thus have the following type:

$$\begin{array}{lll}\mathsf{tp\_lam} & : & \Pi T_1{:}\mathsf{tp}.\, \Pi T_2{:}\mathsf{tp}.\, \Pi E{:}\mathsf{exp} \to \mathsf{exp}.\\ & & \quad (\Pi x{:}\mathsf{exp}.\, \mathsf{of}\; x\; T_1 \to \mathsf{of}\; (E\; x)\; T_2)\\ & & \quad \to \mathsf{of}\; (\mathsf{lam}\; E)\; (\mathsf{arrow}\; T_1\; T_2).\end{array}$$

Representation of a deduction $\mathcal{P}$ with hypotheses Δ requires unique labels for the various hypotheses, in order to return the appropriate variable whenever an hypothesis is used. While we left this correspondence implicit, it should be clear that in the case of $\ulcorner\mathcal{P}'\urcorner$ above, the hypothesis $x{:}\tau_1$ should be considered as labelled by u.

Case:

$$\mathcal{P} = \cfrac{\Delta(x) = \tau}{\Delta \triangleright x : \tau}\,\mathsf{tp\_var}.$$

This case is not represented using a fixed inference rule, but we will have a variable u of type '$\mathsf{of}\; x\, \ulcorner\tau\urcorner$' which implicitly provides a label for the assumption x. We simply return this variable.

$$\ulcorner\mathcal{P}\urcorner \quad = \quad u$$

The adequacy of this representation is now a straightforward exercise, given in the following two properties. We refer to the full signature (which includes the signatures T for Mini-ML types and E for Mini-ML expressions) as TD.

Property 5.7 (Soundness) *If* $\mathcal{P} :: \Delta \triangleright e : \tau$ *then* $\vdash_{TD} \ulcorner\Delta\urcorner$ *Ctx and*

$$\ulcorner\Delta\urcorner \vdash_{TD} \ulcorner\mathcal{P}\urcorner \Uparrow \mathsf{of}\, \ulcorner e\urcorner\, \ulcorner\tau\urcorner.$$

Property 5.8 (Completeness) *Let Δ be a Mini-ML context and $\Gamma = \ulcorner\Delta\urcorner$. If $\vdash_{TD} E \Uparrow \mathsf{exp}$, $\vdash_{TD} T \Uparrow \mathsf{tp}$, and*

$$\Gamma \vdash_{TD} P \Uparrow \mathsf{of}\ E\ T$$

then there exist e, τ, and a derivation $\mathcal{P} :: \Delta \triangleright e : \tau$ such that $\ulcorner e\urcorner = E$, $\ulcorner T\urcorner = \tau$, and $\ulcorner\mathcal{P}\urcorner = P$.

It remains to understand the compositionality property of the bijection. We reconsider the substitution lemma (Lemma 2.4):

If $\Delta \triangleright e_1 : \tau_1$ and $\Delta, x_1{:}\tau_1 \triangleright e_2 : \tau_2$ then $\Delta \triangleright [e_1/x_1]e_2 : \tau_2$.

The proof is by induction on the structure of $\mathcal{P}_2 :: (\Delta, x_1{:}\tau_1 \triangleright e_2 : \tau_2)$. Wherever the assumption $x_1{:}\tau_1$ is used, we substitute a version of the derivation $\mathcal{P}_1 :: \Delta \triangleright e_1 : \tau_1$ where some additional (and unused) typing assumptions may have been added to Δ. A reformulation using the customary notation for hypothetical judgments exposes the similarity to the considerations for the judgment e *Closed* considered in Section 5.1.

$$\text{If}\quad \begin{array}{c} \overline{\triangleright\, x_1{:}\tau_1}\,u_1 \\ \mathcal{P}_2 \\ \triangleright\, e_2{:}\tau_2 \end{array} \quad\text{and}\quad \begin{array}{c} \mathcal{P}_1 \\ \triangleright\, e_1{:}\tau_1 \end{array} \quad\text{then}\quad \begin{array}{c} \mathcal{P}_1 \\ \overline{\triangleright\, e_1{:}\tau_1}\,u_1 \\ [e_1/x_1]\mathcal{P}_2 \\ \triangleright\, [e_1/x_1]e_2 : \tau_2 \end{array}$$

Here, $[e_1/x_1]\mathcal{P}_2$ is the substitution of e_1 for x_1 in the deduction $\mathcal{P}_2$, which is legal since $\mathcal{P}_2$ is a deduction of a judgment parametric in x_1. Furthermore, the deduction $\mathcal{P}_1$ has been substituted for the hypotheses labelled u in $\mathcal{P}_2$, indicated by writing $\mathcal{P}_1$ above the appropriate hypothesis. Using the conventions established for hypothetical and parametric judgments, the final deduction above can also be written as $[\mathcal{P}_1/u_1][e_1/x_1]\mathcal{P}_2$. Compositionality of the representation then requires

$$\begin{array}{rcl} \ulcorner[\mathcal{P}_1/u_1][e_1/x_1]\mathcal{P}_2\urcorner & = & [\ulcorner\mathcal{P}_1\urcorner/u_1][\ulcorner e_1\urcorner/x_1]\ulcorner\mathcal{P}_2\urcorner \\ & \equiv & (\lambda x_1{:}\mathsf{exp}.\ \lambda u_1{:}\mathsf{of}\ x_1\ \ulcorner\tau_1\urcorner.\ \ulcorner\mathcal{P}_2\urcorner)\ \ulcorner e_1\urcorner\ \ulcorner\mathcal{P}_1\urcorner \end{array}$$

After appropriate generalization, this is proved by a straightforward induction over the structure of $\mathcal{P}_2$, just as the substitution lemma for typing derivation which lies at the heart of this property.

Theorem 5.9 (Adequacy) *There is a bijection between deductions*

$$\begin{array}{c} \mathcal{P} \\ x_1{:}\tau_1, \ldots, x_n{:}\tau_n \triangleright e : \tau \end{array}$$

and LF objects P such that

$$x_1{:}\mathsf{exp}, u_1{:}\mathsf{of}\ x_1\ \ulcorner\tau_1\urcorner, \ldots, x_n{:}\mathsf{exp}, u_n{:}\mathsf{of}\ x_n\ \ulcorner\tau_n\urcorner \vdash_{TD} P \Uparrow \mathsf{of}\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner.$$

The bijection is compositional in the sense that for an expression e_i and deduction $\mathcal{P}_i :: \Delta_i \triangleright e_i : \tau_i$ we have

$$\ulcorner[\mathcal{P}_i/u_i][e_i/x_i]\mathcal{P}\urcorner = [\ulcorner\mathcal{P}_i\urcorner/u_i][\ulcorner e_i\urcorner/x_i]\ulcorner\mathcal{P}\urcorner$$

where $\Delta_i = x_1{:}\mathsf{exp}, u_1{:}\mathsf{of}\ x_1\ \ulcorner\tau_1\urcorner, \ldots, x_i{:}\mathsf{exp}, u_i{:}\mathsf{of}\ x_i\ \ulcorner\tau_i\urcorner$.

Proof: As usual, by induction on the given derivations in each direction combined with verifying that correctness of the inverse of the representation function. Compositionality follows by induction on the structure of $\mathcal{P}$. □

5.5 An Elf Program for Mini-ML Type Inference

We now complete the signature from the previous section by transcribing the rules from the previous section and Section 2.5 into Elf. The notation will be suggestive of a reading of this signature as a program for type inference. First, the declarations of Mini-ML types.

```
tp : type.  %name tp T.

nat   : tp.
cross : tp -> tp -> tp.
arrow : tp -> tp -> tp.
```

Next, the typing rules.

```
of : exp -> tp -> type.  %name of P u.
%mode of +E *T.

% Natural Numbers
tp_z     : of z nat.
tp_s     : of (s E) nat
            <- of E nat.
tp_case  : of (case E1 E2 E3) T
            <- of E1 nat
            <- of E2 T
            <- ({x:exp} of x nat -> of (E3 x) T).

% Pairs
tp_pair : of (pair E1 E2) (cross T1 T2)
```

```
           <- of E1 T1
           <- of E2 T2.
tp_fst  : of (fst E) T1
           <- of E (cross T1 T2).
tp_snd  : of (snd E) T2
           <- of E (cross T1 T2).

% Functions
tp_lam : of (lam E) (arrow T1 T2)
          <- ({x:exp} of x T1 -> of (E x) T2).
tp_app : of (app E1 E2) T1
          <- of E1 (arrow T2 T1)
          <- of E2 T2.

% Definitions
tp_letv : of (letv E1 E2) T2
           <- of E1 T1
           <- ({x:exp} of x T1 -> of (E2 x) T2).
tp_letn : of (letn E1 E2) T2
           <- of E1 T1
           <- of (E2 E1) T2.

% Recursion
tp_fix : of (fix E) T
          <- ({x:exp} of x T -> of (E x) T).
```

As for evaluation, we take advantage of compositionality in order to represent substitution of an expression for a bound variable in representation of tp_letn,

$$\frac{\Delta \triangleright e_1 : \tau_1 \qquad \Delta \triangleright [e_1/x]e_2 : \tau_2}{\Delta \triangleright \mathbf{let\,name}\; x = e_1 \;\mathbf{in}\; e_2 : \tau_2}\,\mathsf{tp\_letn}.$$

Since we are using higher-order abstract syntax, e_2 is represented together with its bound variable as a function of type exp $\rightarrow$ exp. Applying this function to the representation of e_1 yields the representation of $[e_1/x]e_2$.

The Elf declarations above are suggestive of an operational interpretation as a program for type inference. The idea is to pose queries of the form `?- of` $\ulcorner e \urcorner$ `T.` where `T` is a free variable subject to instantiation and e is a concrete Mini-ML expression. We begin by considering a simple example: $\mathbf{lam}\; x.\; \langle x, \mathbf{s}\; x\rangle$. For this purpose we assume that `of` has been declared dynamic and `exp` and `tp` are static. This means that free variables of type `exp` and `tp` may appear in an answer.

```
?- of (lam [x] pair x (s x)) T.
```

```
Resolved with clause tp_lam
?- {x:exp} of x T1 -> of (pair x (s x)) T2.
```

In order to perform this first resolution step, the interpreter performed the substitutions

```
E = [x:exp] pair x (s x),
T1 = T1,
T2 = T2,
T = arrow T1 T2.
```

where `E`, `T1`, and `T2` come from the clause `tp_lam`, and `T` appears in the original query. Now the interpreter applies the rules for solving goals of functional type and introduces a new parameter `x`.

```
Introducing new parameter x : exp

x : exp
?- of x T1 -> of (pair x (s x)) T2.
Introducing new parameter u : of x T1.

x : exp,
u : of x T1
?- of (pair x (s x)) T2.
Resolved with clause tp_pair
```

This last resolution again requires some instantiation. We have

```
E1 = x,
E2 = (s x),
T2 = cross T21 T22.
```

Here, `E1`, `E2`, `T21`, and `T22` come from the clause (the latter two renamed from `T1` and `T2`, respectively). Now we have to solve two subgoals, namely `?- of x T21.` and `?- of (s x) T22.` The first subgoal immediately succeeds by using the assumption `u`, which requires the instantiation `T21 = T1` (or vice versa).

```
x : exp,
u : of x T1
?- of x T21.
Resolved with clause u
```

Here is the remainder of the computation.

```
x : exp,
u : of x T1
?- of (s x) T22.
Resolved with clause tp_s
```

This instantiates T22 to nat and produces one subgoal.

```
x : exp,
u : of x T1
?- of x nat.
Resolved with clause u
```

This last step instantiates T1 (and thereby indirectly T21) to nat. Thus we obtain the final answer

```
T = arrow nat (cross nat nat).
```

We can also ask for the typing derivation Q:

```
?- Q : of (lam [x] pair x (s x)) T.

T = arrow nat (cross nat nat),
Q = tp_lam [x:exp] [P:of x nat] tp_pair (tp_s P) P.
```

There will always be at most one answer to a type query, since for each expression constructor there exists at most one applicable clause. Of course, type inference will fail for ill-typed queries, and it will report failure, again because the rules are syntax-directed. We have stated above that there will be at most one answer yet we also know that types of expressions such as **lam** $x.\ x$ are not unique. This apparent contradiction is resolved by noting that the given answer subsumes all others in the sense that all other types will be instances of the given type. This deep property of Mini-ML type inference is called the *principal type property*.

```
?- of (lam [x] x) T.
Resolved with clause tp_lam
?- {x:exp} of x T1 -> of x T2.
Introducing new parameter x
?- of x T1 -> of x T2.
Assuming u1 : of x T1
?- of x T2.
Resolved with clause u1 [with T2 = T1]

T = arrow T1 T1.
```

Here the final answer contains a free variable T1 of type tp. This is legal, since we have declare tp to be a static type. Any instance of the final answer will yield an answer to the original problem and an object of the requested type. This can be expressed by stating that search has constructed a closed object, namely

```
([T1:tp] tp_lam ([x:exp] [P:of x T1] P)) :
{T1:tp} of (lam ([x:exp] x)) (arrow T1 T1).
```

If we interpret this result as a deduction, we see that search has constructed a deduction of a parametric judgment, namely that $\triangleright\ \mathbf{lam}\ x.\ x : \tau_1 \to \tau_1$ for any concrete type τ_1. In order to include such generic derivations we permitted type variables α in our language. The most general or principal derivation above would then be written (in two different notations):

$$\dfrac{\dfrac{}{\triangleright\ x : \alpha}\ u}{\triangleright\ \mathbf{lam}\ x.\ x : \alpha \to \alpha}\ \mathsf{tp\_lam}^{x,u} \qquad \dfrac{\dfrac{}{x{:}\alpha \triangleright x : \alpha}\ \mathsf{tp\_var}}{\triangleright\ \mathbf{lam}\ x.\ x : \alpha \to \alpha}\ \mathsf{tp\_lam}$$

From the program above one can see, that the type inference problem has been reduced to the satisfiability of some equations which arise from the question if a clause head and the goal have a common instance. For example, the goal `?- of (lam [x] x) (cross T1 T2).` will fail immediately, since the only possible rule, `tp_lam`, is not applicable because `arrow T1 T2` and `cross T1 T2` do not have a common instance. The algorithm for finding common instances which also has the additional property that it does not make any unnecessary instantiation is called a *unification algorithm*. For first-order terms (such as LF objects of type `tp` in the type inference problem) a least committed common instance can always be found and is unique (modulo renaming of variables). When variables are allowed to range over functions, this is no longer the case. For example, consider the objects `E2 z` and `pair z z`, where `E2` is a free variable of type `exp -> exp`. Then there are four canonical closed solutions for `E2`:[1]

```
E2 = [x:exp] pair x x ;
E2 = [x:exp] pair z x ;
E2 = [x:exp] pair x z ;
E2 = [x:exp] pair z z.
```

In general, the question whether two objects have a common instance in the LF type theory is undecidable. This follows from the same result (due to Goldfarb [Gol81]) for a much weaker theory, the second-order fragment of the simply-typed lambda-calculus.

The operational reading of LF we sketched so far thus faces a difficulty: one of the basic steps (finding a common instance) is an undecidable problem, and, moreover, may not have a least committed solution. We deal with this problem by approximation: Elf employs an algorithm which finds a greatest common instance or detects failure in many cases and postpones other equations which must be satisfied as *constraints*. In particular, it will solve all problems which are essentially first order, as they arise in the type inference program above. Thus Elf is in spirit a *constraint logic programming language*, even though in many aspects it goes beyond the definition of the CLP family of languages described by Jaffar and Lassez [JL87].

[1] A fifth possibility, `E2 = pair z` is not canonical and η-equivalent to the second solution.

The algorithm, originally discovered by Miller in the simply-typed λ-calculus [Mil91] has been generalized to dependent and polymorphic types in [Pfe91b]. The precise manner in which it is employed in Elf is described in [Pfe91a].[2]. Here we content ourselves with a simple example which illustrates how constraints may arise.

```
?- of (lam [x] x) ((F:tp -> tp) nat).

F = F.
(( arrow T1 T1 = F nat ))
```

Here the remaining constraint is enclosed within double parentheses. Any solution to this equation yields an answer to the original query. It is important to realize that this constitutes only a *conditional success*, that is, we can in general not be sure that the given constraint set is indeed be satisfiable. In the example above, this is obvious: there are infinitely many solutions of which we show two.

```
F = [T:tp] arrow T1 T1,
T1 = T1 ;
F = [T:tp] arrow (arrow T T) (arrow T T),
T1 = arrow nat nat.
```

The same algorithm is also employed during Elf's term reconstruction phase. In practice this means that Elf term reconstruction may also terminate with remaining constraints which, in this case, is considered an error and accompanied by a request to the programmer to supply more type information.

The operational behavior of the program above may not be satisfactory from the point of view of efficiency, since expressions bound to a variable by a **let name** are type-checked once for each occurrence of the variable in the body of the expression. The following is an example for a derivation involving **let name** in Elf.

```
?- Q : of (letn (lam [y] y) ([f] pair (app f z) (app f (pair z z)))) T.
Solving...

T = cross nat (cross nat nat),
Q =
   tp_letn
      (tp_pair
          (tp_app (tp_pair tp_z tp_z)
              (tp_lam [x:exp] [P:of x (cross nat nat)] P))
          (tp_app tp_z (tp_lam [x:exp] [P:of x nat] P)))
      (tp_lam [x:exp] [P:of x T1] P).
More? y
no more solutions
```

[2] *[further discussion of unification elsewhere?]*

Notice the two occurrences of `tp_lam` which means that `(lam [y] y)` was type-checked twice. Usually, ML's type system is defined with explicit constructors for polymorphic types so that we can express $\triangleright$ **lam** $x.\ x : \forall t.\ t \rightarrow t$. The type inference algorithm can then instantiate such a most general type in the body e_2 of a **let name**-expression **let name** $x = e_1$ **in** e_2 without type-checking e_1 again. This is the essence of Milner's algorithm W in [Mil78]. It is difficult to realize this algorithm directly in Elf. Some further discussion and avenues towards a possible solution are given in [Har90], [DP91], and [Lia95]. . Theoretically, however, algorithm W of [Mil78] is not more efficient compared to the algorithm presented above as shown by [Mai92].

5.6 Representing the Proof of Type Preservation

We now return to the proof of type preservation from Section 2.6. In order to prepare for its representation in Elf, we reformulate the theorem to explicitly mention the deductions involved.

For any e, v, τ, $\mathcal{D} :: e \hookrightarrow v$, and $\mathcal{P} :: \triangleright\, e : \tau$ there exists a $\mathcal{Q} :: \triangleright\, v : \tau$.

The proof is by induction on the structure of $\mathcal{D}$ and relies heavily on inversion to predict the shape of $\mathcal{P}$ from the structure of e. The techniques from Section 3.7 suggest casting this proof as a higher-level judgment relating $\mathcal{D}$, $\mathcal{P}$, and $\mathcal{Q}$. This higher-level judgment can be represented in LF and then be implemented in Elf as a type family. We forego the intermediate step and directly map the informal proof into Elf, calling the type family `tps`.

```
tps : eval E V -> of E T -> of V T -> type.
%mode tps +D +P -Q.
```

All of the cases in the induction proof now have a direct representation in Elf. The interesting cases involve appeals to the substitution lemma (Lemma 2.4).

Case: $\mathcal{D} = \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z}.$

Then we have to show that for any type τ such that $\triangleright\, \mathbf{z} : \tau$ is derivable, $\triangleright\, \mathbf{z} : \tau$ is derivable. This is obvious.

There are actually two slightly different, but equivalent realizations of this case. The first uses the deduction $\mathcal{P} :: \triangleright\, \mathbf{z} : \tau$ that exists by assumption.

```
tps_z0 : tps (ev_z) P P.
```

The second, which we prefer, uses inversion to conclude that $\mathcal{P}$ must be `tp_z`, since it is the only rule which assigns a type to $\mathbf{z}$.

```
tps_z     : tps (ev_z) (tp_z) (tp_z).
```

The appeal to the inversion principle is implicit in these declarations. For each $\mathcal{D}$ and $\mathcal{P}$ there should be a $\mathcal{Q}$ such that tps $\ulcorner\mathcal{D}\urcorner$ $\ulcorner\mathcal{P}\urcorner$ $\ulcorner\mathcal{Q}\urcorner$ is inhabited. The declaration above appears to work only for the case where the second argument $\mathcal{P}$ is the axiom tp_z. But by inversion we know that this is the only possible case. This pattern of reasoning is applied frequently when representing proofs of meta-theorems.

The next case deals with the successor constructor for natural numbers. We have taken the liberty of giving names to the deductions whose existence is shown in the proof in Section 2.6. This will help use to connect the informal statement with its implementation in Elf.

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e_1 \hookrightarrow v_1\end{matrix}}{\mathbf{s}\ e_1 \hookrightarrow \mathbf{s}\ v_1}\,\mathsf{ev\_s}$. Then

$\mathcal{P} :: \triangleright\, \mathbf{s}\ e_1 : \tau$	By assumption
$\mathcal{P}_1 :: \triangleright\, e_1 : \mathbf{nat}$ and $\tau = \mathbf{nat}$	By inversion
$\mathcal{Q}_1 :: \triangleright\, v_1 : \mathbf{nat}$	By ind. hyp. on $\mathcal{D}_1$
$\mathcal{Q} :: \triangleright\, \mathbf{s}\ v_1 : \mathbf{nat}$	By rule tp_s

Recall that an appeal to the induction hypothesis is modelled by a recursive call in the program which implements the proof. Here, the induction hypothesis is applied to $\mathcal{D}_1 :: e_1 \hookrightarrow v_1$ and $\mathcal{P}_1 :: \triangleright\, e_1 : \mathbf{nat}$ to conclude that there is a $\mathcal{Q}_1 :: \triangleright\, v_1 : \mathbf{nat}$. This is what we needed to show and can thus be directly returned.

```
tps_s     : tps (ev_s D1) (tp_s P1) (tp_s Q1)
             <- tps D1 P1 Q1.
```

We return to the cases involving **case**-expressions later after we have discussed the case for functions. The rules for pairs are straightforward.

```
tps_pair : tps (ev_pair D2 D1) (tp_pair P2 P1) (tp_pair Q2 Q1)
             <- tps D1 P1 Q1
             <- tps D2 P2 Q2.
tps_fst  : tps (ev_fst D1) (tp_fst P1) Q1
             <- tps D1 P1 (tp_pair Q2 Q1).
tps_snd  : tps (ev_snd D1) (tp_snd P1) Q2
             <- tps D1 P1 (tp_pair Q2 Q1).
```

There is an important phenomenon one should note here. Since we used the backwards arrow notation in the declarations for ev_pair and tp_pair

```
ev_pair : eval (pair E1 E2) (pair V1 V2)
            <- eval E1 V1
            <- eval E2 V2.

tp_pair : of (pair E1 E2) (cross T1 T2)
          <- of E1 T1
          <- of E2 T2.
```

their arguments are reversed from what one might expect. This is why we called the first argument to `ev_pair` above `D2` and the second argument `D1`, and similiary for `tp_pair`. The case for **lam**-expressions is simple, since they evaluate to themselves. For stylistic reasons we apply inversion here as in all other cases.

```
tps_lam     : tps (ev_lam) (tp_lam P) (tp_lam P).
```

The case for evaluating an application $e_1\ e_2$ is more complicated than the cases above. The informal proof appeals to the substitution lemma.

Case: $\mathcal{D} = \dfrac{\begin{array}{ccc} \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1' & e_2 \hookrightarrow v_2 & [v_2/x]e_1' \hookrightarrow v \end{array}}{e_1\ e_2 \hookrightarrow v}\ \mathsf{ev\_app}.$

$\mathcal{P} :: \triangleright\ e_1\ e_2 : \tau_1$	By assumption
$\mathcal{P}_1 :: \triangleright\ e_1 : \tau_2 \to \tau_1$ and $\mathcal{P}_2 :: \triangleright\ e_2 : \tau_2$ for some τ_2	By inversion
$\mathcal{Q}_1 :: \triangleright\ \mathbf{lam}\ x.\ e_1' : \tau_2 \to \tau_1$	By ind. hyp. on $\mathcal{D}_1$
$\mathcal{Q}_1' :: x{:}\tau_2 \triangleright e_1' : \tau_1$	By inversion
$\mathcal{Q}_2 :: \triangleright\ v_2 : \tau_2$	By ind. hyp. on $\mathcal{D}_2$
$\mathcal{P}_3 :: \triangleright\ [v_2/x]e_1' : \tau_1$	By the Substitution Lemma 2.4
$\mathcal{Q}_3 :: \triangleright\ v : \tau_1$	By ind. hyp. on $\mathcal{D}_3$

We repeat the declarations of the `ev_app` and `tp_lam` clauses here with some variables renamed in order to simplify the correspondence to the names used above.

```
ev_lam : eval (app E1 E2) V
          <- eval E1 (lam E1')
          <- eval E2 V2
          <- eval (E1' V2) V.

tp_lam : of (lam E1') (arrow T2 T1)
          <- ({x:exp} of x T2 -> of (E1' x) T1).
```

The deduction $\mathcal{Q}_1$ is a deduction of a parametric and hypothetical judgment (parametric in x, hypothetical in $\triangleright\ x{:}\tau_2$). In Elf this is represented as a function which, when applied to `V2` and an object `Q2 : of V2 T2` yields an object of type `of (E1' V2) T1`, that is

```
Q1' : {x:exp} of x T2 -> of (E1' x) T1.
```

The Elf variable `E1' : exp -> exp` represents λx:exp. $\ulcorner e_1' \urcorner$, and `V2` represents v_2. Thus the appeal to the substitution lemma has been transformed into a function application using `Q1'`, that is, `P3 = Q1' V2 Q2`.

```
tps_app : tps (ev_app D3 D2 D1) (tp_app P2 P1) Q3
           <- tps D1 P1 (tp_lam Q1')
           <- tps D2 P2 Q2
           <- tps D3 (Q1' V2 Q2) Q3.
```

This may seem like black magic—where did the appeal to the substitution lemma go? The answer is that it is hidden in the proof of the adequacy theorem for the representation of typing derivations (Theorem 5.9) combined with the substitution lemma for LF itself! We have thus factored the proof effort: in the proof of the adequacy theorem, we establish that the typing judgment employs parametric and hypothetical judgments (which permit weakening and substitution). The implementation above can then take this for granted and model an appeal to the substitution lemma simply by function application.

One very nice property is the conciseness of the representation of the proofs of the meta-theorems in this fashion. Each case in the induction proof is represented directly as a clause, avoiding explicit formulation and proof of many properties of substitution, variable occurrences, *etc.* This is due to the principles of higher-order abstract syntax, judgments-as-types, and hypothetical judgments as functions. Another important factor is the Elf type reconstruction algorithm which eliminates the need for much redundant information. In the clause above, for example, we need to refer explicitly to only one expression (the variable `V2`). All other constraints imposed on applications of inferences rules can be inferred in a most general way in all of these examples. To illustrate this point, here is the fully explicit form of the above declaration, as generated by Elf's term reconstruction.

```
tps_app :
  {E:exp -> exp} {V2:exp} {E1:exp} {T:tp} {D3:eval (E V2) E1}
     {T1:tp} {Q1':{E1:exp} of E1 T1 -> of (E E1) T} {Q2:of V2 T1}
     {Q3:of E1 T} {E2:exp} {D2:eval E2 V2} {P2:of E2 T1} {E3:exp}
     {D1:eval E3 (lam E)} {P1:of E3 (arrow T1 T)}
     tps (E V2) E1 T D3 (Q1' V2 Q2) Q3 -> tps E2 V2 T1 D2 P2 Q2
        -> tps E3 (lam E) (arrow T1 T) D1 P1 (tp_lam T1 E T Q1')
        -> tps (app E3 E2) E1 T (ev_app E V2 E1 E2 E3 D3 D2 D1)
               (tp_app E2 T1 E3 T P2 P1) Q3.
```

We skip the two cases for **case**-expressions and only show their implementation. The techniques we need have all been introduced.

```
tps_case_z: tps (ev_case_z D2 D1) (tp_case P3 P2 P1) Q2
             <- tps D2 P2 Q2.

tps_case_s: tps (ev_case_s D3 D1) (tp_case P3 P2 P1) Q3
             <- tps D1 P1 (tp_s Q1')
             <- tps D3 (P3 V1 Q1') Q3.
```

Next, we come to the cases for definitions. For **let val**-expressions, no new considerations arise.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ e_1 \hookrightarrow v_1 & [v_1/x]e_2 \hookrightarrow v \end{array}}{\textbf{let val}\ x = e_1\ \textbf{in}\ e_2 \hookrightarrow v}\ \textsf{ev\_letv}.$

$\mathcal{P} :: \triangleright\ \textbf{let val}\ x = e_1\ \textbf{in}\ e_2 : \tau$	By assumption
$\mathcal{P}_1 :: \triangleright\ e_1 : \tau_1$ and	
$\mathcal{P}_2 :: x{:}\tau_1 \triangleright e_2 : \tau$ for some τ_1	By inversion
$\mathcal{Q}_1 :: \triangleright\ v_1 : \tau_1$	By ind. hyp. on $\mathcal{D}_1$
$\mathcal{P}_2' :: \triangleright\ [v_1/x]e_2 : \tau$	By the Substitution Lemma 2.4
$\mathcal{Q}_2 :: \triangleright\ v : \tau$	By ind. hyp. on $\mathcal{D}_2$

```
tps_letv : tps (ev_letv D2 D1) (tp_letv P2 P1) Q2
            <- tps D1 P1 Q1
            <- tps D2 (P2 V1 Q1) Q2.
```

let name-expressions may at first sight appear to be the most complicated case. However, the substitution at the level of expressions is dealt with via compositionality as in evaluation, so the representation of this case is actually quite simple.

Case: $\mathcal{D} = \dfrac{\begin{array}{c} \mathcal{D}_2 \\ [e_1/x]e_2 \hookrightarrow v \end{array}}{\textbf{let name}\ x = e_1\ \textbf{in}\ e_2 \hookrightarrow v}\ \textsf{ev\_letn}.$

$\mathcal{P} :: \triangleright\ \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 : \tau$	By assumption
$\mathcal{P}_2 :: \triangleright\ [e_1/x]e_2 : \tau$	By inversion
$\mathcal{Q}_2 :: \triangleright\ v : \tau$	By ind. hyp. on $\mathcal{D}_2$

```
tps_letn : tps (ev_letn D2) (tp_letn P2 P1) Q2
            <- tps D2 P2 Q2.
```

The case of fixpoint follows the same general pattern as the case for application in that we need to appeal to the substitution lemma. The solution is analogous.

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ [\textbf{fix}\ x.\ e_1/x]e_1 \hookrightarrow v\end{matrix}}{\textbf{fix}\ x.\ e_1 \hookrightarrow v}\ \mathsf{ev\_fix}.$

$\mathcal{P} :: \triangleright\ \textbf{fix}\ x.\ e_1 : \tau$	By assumption
$\mathcal{P}_1 :: x : \tau \triangleright e_1 : \tau$	By inversion
$\mathcal{P}_1' :: \triangleright\ [\textbf{fix}\ x.\ e_1/x]e_1 : \tau$	By the Substitution Lemma 2.4
$\mathcal{Q}_1 :: \triangleright\ v : \tau$	By ind. hyp. on $\mathcal{D}_1$

In the representation,

```
P1 : {x:exp} of x T -> of (E1 x) T
```

and thus

```
(P1 (fix E1)) : of (fix E1) T -> of (E1 (fix E1)) T
```

and

```
(P1 (fix E1) (tp_fix P1)) : of (E1 (fix E1)) T
```

This is the representation of the deduction $\mathcal{P}_1'$, since

$$\ulcorner[\textbf{fix}\ x.\ e_1/x]e_1\urcorner = [\ulcorner\textbf{fix}\ x.\ e_1\urcorner/x]\ulcorner e_1\urcorner \equiv (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1\urcorner)\ (\mathsf{fix}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1\urcorner)).$$

```
tps_fix : tps (ev_fix D1) (tp_fix P1) Q1
           <- tps D1 (P1 (fix E1) (tp_fix P1)) Q1.
```

Here is a simple example which illustrates the use of the `tps` type family as a program. First, we abbreviate the expression

$$\textbf{let name}\ f = \textbf{lam}\ x.\ x\ \textbf{in let}\ g = f\ f\ \textbf{in}\ g\ g$$

by `e0`, using the definitional mechanism of Elf. The we generate the typing derivation with a `%solve` declaration and call it `p0`. Next we evaluate `e0` and call the evaluation `d0`. Then we pose a query that will execute the proof of type preservation to generate a substitution for `Q`, the typing derivation for the value of the expression above.

```
e0 : exp = letn (lam [x] x) ([f] letn (app f f) ([g] app g g)).
%solve p0 : of e0 T.
%solve d0 : eval e0 V.
%query 1 * tps d0 p0 Q.
```

Among other information, this will print

```
Q = tp_lam ([x:exp] [u:of x T1] u).
```

Of course, this is a very indirect way to generate a typing derivation of **lam** $x.\ x$, but illustrates the computational content of the type family `tps` we defined.

5.7 Exercises

Exercise 5.1 Carry out three representative cases in the proof of Property 5.4. Where do we require the assumption that Γ must be of a certain form? Construct a counterexample which shows the falsehood of careless generalization of the theorem to admit arbitary contexts Γ.

Exercise 5.2 Carry out three representative cases in the proof of the Adequacy Theorem 5.5.

Exercise 5.3 Modify the natural semantics for Mini-ML such that only closed λ-expressions have a value. How does this affect the proof of type preservation?

Exercise 5.4 Write Elf programs

1. to count the number of occurrences of bound variables in a Mini-ML expression;
2. to remove all vacuous **let**-bindings from a Mini-ML expression;
3. to rewrite all occurrences of expressions of the form $(\mathbf{lam}\ x.\ e_2)\ e_1$ to $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$.

Exercise 5.5 For each of the following statements, prove them informally and represent the proof in Elf, or give a counterexample if the statement is false.

1. For any expressions e_1 and e_2, evaluation of $(\mathbf{lam}\ x.\ e_2)\ e_1$ yields a value v if and only if evaluation of $\mathbf{let\,val}\ x = e_1\ \mathbf{in}\ e_2$ yields v.
2. For any expressions e_1 and e_2, evaluation of $(\mathbf{lam}\ x.\ e_2)\ e_1$ yields a value v if and only if evaluation of $\mathbf{let\,name}\ x = e_1\ \mathbf{in}\ e_2$ yields v.
3. For *values* v_1, the expression $(\mathbf{lam}\ x.\ e_2)\ v_1$ has type τ if and only if the expression $\mathbf{let\,val}\ x = v_1\ \mathbf{in}\ e_2$ has type τ.
4. For *values* v_1, the expression $(\mathbf{lam}\ x.\ e_2)\ v_1$ has type τ if and only if the expression $\mathbf{let\,name}\ x = v_1\ \mathbf{in}\ e_2$ has type τ.
5. Evaluation is deterministic, that is, whenever $e \hookrightarrow v_1$ and $e \hookrightarrow v_2$ then $v_1 = v_2$ (modulo renaming of bound variables, as usual).

Exercise 5.6 Give an LF representation of the fragment of Mini-ML which includes pairing, first and second projection, functions and application, and definitions with **let val** without using hypothetical judgments. Thus the typing judgment should be represented as a ternary type family, say, `hastype`, indexed by a representation of the context Δ and representations of e and τ. We would then look for a representation function $\ulcorner\cdot\urcorner$ which satisfies

$$\Gamma \vdash \ulcorner\mathcal{P}\urcorner : \mathsf{hastype}\ \ulcorner\Delta\urcorner\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner$$

for a suitable Γ, whenever $\mathcal{P}$ is a valid deduction of $\Delta \triangleright e : \tau$.

Exercise 5.7 Illustrate by means of an example why declaring the type `tp` as dynamic might lead to undesirable backtracking and unexpected answers during type inference for Mini-ML with the program in Section 5.5. Can you construct a situation where the program diverges on a well-typed Mini-ML expression? How about on a Mini-ML expression which is not well-typed?

Exercise 5.8 Extend the implementation of the Mini-ML interpreter, type inference, and proof of type preservation to include

1. unit, void, and disjoint sum types (see Exercise 2.7),
2. lists (see Exercise 2.8).

Exercise 5.9 Consider the call-by-name version of Mini-ML with lazy constructors as sketched in Exercise 2.13. Recall that neither the arguments to functions, nor the arguments to constructors ($\mathbf{s}$ and $\langle\cdot,\cdot\rangle$) should be evaluated.

1. Implement an interpreter for the language and show a few expressions that highlight the differences in the operational semantics.
2. Implement type inference.
3. Define and implement a suitable notion of value.
4. Prove value soundness and implement your proof.
5. Prove type preservation and implement your proof.

Discuss the main differences between the development for Mini-ML and its call-by-name variant.

Exercise 5.10 The definition of the judgment e *Closed* follows systematically from the representation of expressions in higher-order abstract syntax, because object-level variables are represented by meta-level variables. This exercise explores a generalization of this fact. Assume we have a signature Σ_0 in the simply-typed lambda-calculus that declares exactly one type constant a and some unspecified number of object constants $c_1, \ldots, c_n$. Define an LF signature Σ_1 that extends Σ_0 by a new family

$$\mathsf{closed} \ : \ a \to \mathsf{type}$$

such that

$$\vdash_{\Sigma_0} N : a$$

if and only if

$$\Gamma \vdash_{\Sigma_1} N : a \quad \text{and} \quad \Gamma \vdash_{\Sigma_1} M : \mathsf{closed}\ N$$

provided Γ no has declaration of the form $u{:}\Pi y_1{:}A_1 \ldots \Pi y_m{:}A_m.\ \mathsf{closed}\ P$.

Exercise 5.11 Write Elf programs to determine if a Mini-ML expression is free of the recursion operation **fix** and at the same time

1. linear (every bound variable occurs exactly once);
2. affine (every bound variable occurs at most once);
3. relevant (every bound variable occurs at least once).

Since only one branch in a case statement will be taken during evaluation, a bound variable must occur exactly once in each branch in a linear expression, may occur at most once in each branch in an affine expression, and must occur at least once in each branch in a relevant expression.

Exercise 5.12 Instead of substituting in the typing rule for **let name**-expressions we could extend contexts to record the definitions for variables bound with **let name**.

$$\text{Contexts} \quad \Delta \quad ::= \quad \cdot \mid \Delta, x{:}\tau \mid \Delta, x = e$$

Variables must still occur at most once in a context (no variable may be declared and defined). We would replace the rule tp_letn by the following two rules.

$$\frac{\Delta \rhd e_1 : \tau_1 \qquad \Delta, x = e_1 \rhd e_2 : \tau_1}{\Delta \rhd \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\,\mathsf{tp\_letn0} \qquad \frac{x = e \text{ in } \Delta \qquad \Delta \rhd e : \tau}{\Delta \rhd x : \tau}\,\mathsf{tp\_var0}$$

There are at least two ways we can view this modification for representation in the framework.

1. We use a new judgment, $x = e$, which is introduced only as a hypothesis into a derivation.
2. We view a hypothesis $x = e$ as the assumption of an *inference rule*. We might write this as

$$\frac{\rhd e_1 : \tau_1 \qquad \begin{array}{c} \dfrac{\rhd e_1 : \tau}{\rhd x : \tau}\,u_\tau \\ \vdots \\ \rhd e_2 : \tau_2 \end{array}}{\rhd \textbf{let name}\ x = e_1\ \textbf{in}\ e_2}\,\mathsf{tp\_let}^{x,u}.$$

The subscript τ in the hypothetical rule u indicates that in each application of u we may choose a different type τ. Hypothetical rules have been investigated by Schroeder-Heister [SH84].

1. Show the proper generalization and the new cases in the informal proof of type preservation using rules tp_letn0 and tp_var0.
2. Give the Elf implementation of type inference using alternative 1.
3. Implement the modified proof of type preservation in Elf using alternative 1.
4. Give the Elf implementation of type inference using alternative 2.
5. Implement the modified proof of type preservation in Elf using alternative 2.

Exercise 5.13 [*on the value restriction or its absence*]

Exercise 5.14 [*on interpreting let name as let value, connect to value restriction*]

Exercise 5.15 The typing rules for Mini-ML in Section 2.5 are not a realistic basis for an implementation, since they require e_1 in an expression of the form $\mathbf{let\,name}\ u = e_1\ \mathbf{in}\ e_2$ to be re-checked at every occurrence of u in e_2. This is because we may need to assign different types to e_1 for different occurrences of u.

Fortunately, all the different types for an expression e can be seen as instances of a most general *type schema* for e. In this exercise we explore an alternative formulation of Mini-ML which uses explicit type schemas.

$$\begin{array}{rccl} \text{Types} & \tau & ::= & \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \alpha \\ \text{Type Schemas} & \sigma & ::= & \tau \mid \forall \alpha.\, \sigma \end{array}$$

Type schemas σ are related to types τ through *instantiation*, written as $\sigma \preceq \tau$. This judgment is defined by

$$\frac{}{\tau \preceq \tau}\,\mathsf{inst\_tp} \qquad \frac{[\tau'/\alpha]\sigma \preceq \tau}{\forall \alpha.\, \sigma \preceq \tau}\,\mathsf{inst\_all}.$$

We modify the judgment $\Delta \triangleright e : \tau$ and add a second judgment, $\Delta \vartriangleright\!\!\vartriangleright e : \sigma$ stating that e has type schema σ. The typing rule for **let name** now no longer employs substitution, but refers to a schematic type for the definition. It must therefore be possible to assign type schemas to variables which are instantiated when we need an actual type for a variable.

$$\frac{\Delta \vartriangleright\!\!\vartriangleright e_1 : \sigma_1 \qquad \Delta, x{:}\sigma_1 \triangleright e_2 : \tau_2}{\Delta \triangleright \mathbf{let\,name}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\,\mathsf{tp\_letn} \qquad \frac{\Delta(x) = \sigma \qquad \sigma \preceq \tau}{\Delta \triangleright x : \tau}\,\mathsf{tp\_var}$$

Type schemas can be derived for expressions by means of quantifying over free type variables.

$$\frac{\Delta \triangleright e : \tau}{\Delta \mathrel{\triangleright\!\!\triangleright} e : \tau}\,\mathsf{tpsc\_tp} \qquad \frac{\Delta \mathrel{\triangleright\!\!\triangleright} e : \sigma}{\Delta \mathrel{\triangleright\!\!\triangleright} e : \forall\alpha.\,\sigma}\,\mathsf{tpsc\_all}^{\alpha}$$

Here the premiss of the $\mathsf{tpsc\_all}^{\alpha}$ rule must be parametric in α, that is, α must not occur free in the context Δ.

In the proofs and implementations below you may restrict yourself to the fragment of the language with functions and **let name**, since the changes are orthogonal to the other constructs of the language.

1. Give an example which shows why the restriction on the tpsc_all rule is necessary.

2. Prove type preservation for this formulation of Mini-ML. Carefully write out and prove any substitution lemmas you might need, but you may take weakening and exchange for granted.

3. State the theorem which asserts the equivalence of the new typing rules when compared to the formulation in Section 2.5.

4. Prove the easy direction of the theorem in item 3. Can you conjecture the critical lemma for the opposite direction?

5. Implement type schemas, schematic instantiation, and the new typing judgments in Elf.

6. Unlike our first implementation, the new typing rules do not directly provide an implementation of type inference for Mini-ML in Elf. Show the difficulty by means of an example.

7. Implement the proof of type preservation from item 2 in Elf.

8. Implement one direction of the equivalence proof from item 3 in Elf.

Exercise 5.16 [*about the Milner-Mycroft calculus with explicit types for polymorphic let and recursion*]

Bibliography

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS theorem proving system. In Ewing Lusk and Russ Overbeek, editors, *9th International Conference on Automated Deduction*, pages 760–761, Argonne, Illinois, May 1988. Springer-Verlag LNCS 310. System abstract.

[All75] William Allingham. *In Fairy Land.* Longmans, Green, and Co., London, England, 1875.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1933.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

[dB68] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM Press, 1982.

[Dow93] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 139–145, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[DP91] Scott Dietzen and Frank Pfenning. A declarative alternative to assert in logic programming. In Vijay Saraswat and Kazunori Ueda, editors, *International Logic Programming Symposium*, pages 372–386. MIT Press, October 1991.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 121–136, Chapel Hill, North Carolina, April 1989. Springer-Verlag LNCS 355.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Geu92] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992.

[Han91] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, January 1991. Available as Technical Report MS-CIS-91-09.

[Han93] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[HB34] David Hilbert and Paul Bernays. *Grundlagen der Mathematik.* Springer-Verlag, Berlin, 1934.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM89] John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[Hue73] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.

[Lia95] Chuck Liang. *Object-Level Substitutions, Unification and Generalization in Meta Logic*. PhD thesis, University of Pennsylvania, December 1995.

[Mai92] H.G. Mairson. Quantifier elimination and parametric polymorphism in programming languages. *Journal of Functional Programming*, 2(2):213–226, April 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[ML85] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[New65] Allen Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. E. Taulbee, editors, *Electronic Information Handling*, pages 195–208, Washington, D.C., 1965. Spartan Books.

[NGdV94] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[PW91] David Pym and Lincoln A. Wallen. Proof search in the $\lambda\Pi$-calculus. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 309–340. Cambridge University Press, 1991.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[Sal90] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, France, May 1990.

[Sch00] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[SH84] Peter Schroeder-Heister. A natural extension of natural deduction. *The Journal of Symbolic Logic*, 49(4):1284–1300, December 1984.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.