from a domain with a decidable equality theory. This appears to be a reasonable compromise that can make the expressive power of dependent types available to the programmer without sacrificing decidable and efficient type-checking.

7.3 Logical Frameworks

In the previous section, we illustrated how (linear) type theory could be used as the foundation of (linear) functional programming. The demands of a complete functional language and, in particular, the presence of data types and recursion makes it difficult to attain decidable type-checking. In this section we discuss another application of linear type theory as the foundation for a *logical framework*.

A logical framework is a meta-language for the specification and implementation of deductive systems. This includes applications in various logics and programming languages For surveys of logical frameworks and their applications, see [Pfe96, BM01, Pfe01].

One of the most expressive current frameworks is LF [HHP93] which is based on a λ-calculus with dependent types. There are many elegant encodings of deductive systems in such a framework that can be found in the references above. However, there are also many systems occurring both in logic and programming languages, for which encodings are awkward. Examples are substructural logic (such as linear logic), calculi for concurrency (such as the π-calculus), or programming languages with state (such as Mini-ML with mutable references). Even for pure languages such as Haskell, some aspects of the operational semantics such as laziness and memoization are difficult to handle. This is because at a lower level of abstraction, the implementations of even the purest languages are essentially state-based.

In response to these shortcomings, a linear logical framework (LLF) has been developed [CP96, Cer96, CP98]. This framework solved some of the problems mentioned above. In particular, it allows natural encodings of systems with state. However, it did not succeed with respect to intrinsic notions of concurrency. We will try to give an intuitive explanation of why this is so after showing what the system looks like.

First, in a nutshell, the system LF. The syntax has the following form:

$$\begin{array}{lrcl} \text{Types} & A & ::= & a\, M_1 \ldots M_n \mid \Pi x{:}A_1 \ldots A_2 \\ \text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\, M \mid M_1\, M_2 \end{array}$$

Here, a ranges over *type families* indexed by object M_i, c ranger over *object constants*. Unlike the functional language above, these are not declared by formation rules, introduction rules, and elimination rules. In fact, we must avoid additional introduction and elimination rules because the corresponding local reductions make the equational theory (and therefore type checking) difficult to manage.

Instead, all judgments will be *parametric* in these constants! They are declared in a *signature* Σ which is a global analogue of the context Γ. We need kinds in order to classify type families.

$$\begin{array}{rrcl} \text{Kinds} & K & ::= & \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ \text{type} \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \end{array}$$

As a simple example, we consider a small imperative language. We distinguish expressions (which have a value, but no effect) from commands (which have an effect, but no value). Expressions are left unspecified, except that variables are allowed as expressions and that we must have boolean values true and false. In other words, our example is parametric in the language of expressions; the only requirements are that they cannot have an effect, and that they must allow variables. Commands are no-ops (skip), sequential composition $(C_1; C_2)$, parallel composition $(C_1 \parallel C_2)$, assignment $(x := e)$ and the allocation of a new local variable $(\mathbf{new}\, x.\ C)$. We also have a conditional construct $(\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)$ and a loop $\mathbf{loop}\, l.\ C$.

$$\begin{array}{rrcl} \text{Expression Types} & \tau & ::= & \mathbf{bool} \mid \ldots \\ \text{Expressions} & e & ::= & x \mid \mathbf{true} \mid \mathbf{false} \mid \ldots \\ \text{Commands} & C & ::= & \mathbf{skip} \mid C_1; C_2 \mid C_1 \parallel C_2 \mid x := e \mid \mathbf{new}\, x{:}\tau.\ C \\ & & & \mid \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{loop}\, l.\ C \end{array}$$

For the loop construct $\mathbf{loop}\, l.\ C$, we introduce a label l with scope C. This label is considered a new command in C, where invoking l corresponds to a copy of the loop body. Thus executing $\mathbf{loop}\, l.\ C$ reduces to executing $[\mathbf{loop}\, l.\ C/l]C$. Note that this only allows backwards jumps and does not model a general **goto**. Its semantics is also different from goto if it is not the "last" command in a loop (see Exercise **??** where you are also asked to model a **while**-loop using **loop**). The simplest infinite loop is $\mathbf{loop}\, l.\ l$.

We now turn to the representation of syntax. We have a framework type tp representing object language types, and index expressions by their object language type. We show the representation function $\ulcorner\tau\urcorner$ and $\ulcorner e\urcorner$.

$$\begin{array}{lcllcl} & & & \mathsf{tp} & : & \text{type} \\ \ulcorner\mathbf{bool}\urcorner & = & \mathsf{bool} & \mathsf{bool} & : & \mathsf{tp} \\ & & & \mathsf{var} & : & \mathsf{tp} \to \text{type} \\ & & & \mathsf{exp} & : & \mathsf{tp} \to \text{type} \\ \ulcorner x\urcorner & = & \mathsf{v}\,\ulcorner\tau\urcorner\, x & \mathsf{v} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathsf{exp}(t) \\ \ulcorner\mathbf{true}\urcorner & = & \mathsf{true} & \mathsf{true} & : & \mathsf{exp}(\mathsf{bool}) \\ \ulcorner\mathbf{false}\urcorner & = & \mathsf{false} & \mathsf{false} & : & \mathsf{exp}(\mathsf{bool}) \end{array}$$

One of the main points to note here is that variables of our imperative language are represented by variables in the framework. For the sake of convenience, we choose the same name for a variable and its representation. The type of such a variable in the framework will be $\mathsf{var}(\ulcorner\tau\urcorner)$ depending on its declared type. v is the coercion from a variable of type τ to an expression of type τ. It has to be indexed by a type t, because it could be applied to an expression of any type.

Commands on the other hand do not return a value, so their type is not indexed. In order to maintain the idea the variables are represented by variables, binding constructs in the object language (namely **new** and **loop**) must be translated so they bind the corresponding variable in the meta-language. This idea is called *higher-order abstract syntax* [PE88], since the type of such constructs in generally of order 2 (the constructors new and loop take functions as arguments).

$$
\begin{array}{lcl}
\ulcorner \mathbf{skip} \urcorner & = & \mathsf{skip} \\
\ulcorner C_1 ; C_2 \urcorner & = & \mathsf{seq}\,\ulcorner C_1 \urcorner\,\ulcorner C_2 \urcorner \\
\ulcorner C_1 \parallel C_2 \urcorner & = & \mathsf{par}\,\ulcorner C_1 \urcorner\,\ulcorner C_2 \urcorner \\
\ulcorner x := e \urcorner & = & \mathsf{assign}\,\ulcorner \tau \urcorner\, x\,\ulcorner e \urcorner \quad \text{for } x \text{ and } e \text{ of type } \tau \\
\ulcorner \mathbf{new}\, x{:}\tau.\, C \urcorner & = & \mathsf{new}\,\ulcorner \tau \urcorner\,(\lambda x{:}\mathsf{var}(\ulcorner \tau \urcorner).\,\ulcorner C \urcorner) \\
\ulcorner \mathbf{if}\; e\; \mathbf{then}\; C_1\; \mathbf{else}\; C_2 \urcorner & = & \mathsf{if}\,\ulcorner e \urcorner\,\ulcorner C_1 \urcorner\,\ulcorner C_2 \urcorner \quad \text{for } e \text{ of type } \mathbf{bool} \\
\ulcorner \mathbf{loop}\, l.\, C \urcorner & = & \mathsf{loop}\;(\lambda l{:}\mathsf{cmd}.\,\ulcorner C \urcorner)
\end{array}
$$

In the declarations below, wee see that assign and new need to be indexed by their type, and that a conditional command branches depending on a boolean expression. In that way only well-typed commands can be represented—others will be rejected as ill-typed in the framework.

$$
\begin{array}{lcl}
\mathsf{cmd} & : & \mathrm{type} \\
\mathsf{skip} & : & \mathsf{cmd} \\
\mathsf{seq} & : & \mathsf{cmd} \to \mathsf{cmd} \to \mathsf{cmd} \\
\mathsf{par} & : & \mathsf{cmd} \to \mathsf{cmd} \to \mathsf{cmd} \\
\mathsf{assign} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathsf{exp}(t) \to \mathsf{cmd} \\
\mathsf{new} & : & \forall t{:}\mathsf{tp}.\ (\mathsf{var}(t) \to \mathsf{cmd}) \to \mathsf{cmd} \\
\mathsf{if} & : & \mathsf{exp}(\mathsf{bool}) \to \mathsf{cmd} \to \mathsf{cmd} \\
\mathsf{loop} & : & (\mathsf{cmd} \to \mathsf{cmd}) \to \mathsf{cmd}
\end{array}
$$

So far, we have not needed linearity. Indeed, the declarations above are just a standard means of representing syntax in LF using the expressive power of dependent types for data representation.

Next we would like to represent the operational semantics in the style of encoding that we have used in this class before, beginning with expressions. We assume that for each variable x of type τ we have an affine assumption $x = v$ for a value v of type τ. In a context $x_1 = v_1, \ldots, x_n = v_n$ all variables x_i must be distinct. In judgmental notation, our judgment would be $\Psi \vdash^a e \hookrightarrow v$, where Ψ represents the store and $\vdash^a$ is an affine hypothetical judgment. In our fragment it would be represented by only three rules

$$\frac{}{\Psi, x = v \vdash^a x \hookrightarrow v}\ \mathsf{ev\_var}$$

$$\frac{}{\Psi \vdash^a \mathbf{true} \hookrightarrow \mathbf{true}}\ \mathsf{ev\_true} \qquad \frac{}{\Psi \vdash^a \mathbf{false} \hookrightarrow \mathbf{false}}\ \mathsf{ev\_false}$$

Since our framework is linear and not affine, we need to take care of eliminating unconsumed hypotheses. We have

$$\begin{array}{rcl} \ulcorner x = v \urcorner & = & \mathsf{value}\,\ulcorner\tau\urcorner\,x\,\ulcorner v\urcorner \\ \ulcorner e \hookrightarrow v \urcorner & = & \mathsf{eval}\,\ulcorner\tau\urcorner\,\ulcorner e\urcorner\,\ulcorner t\urcorner \end{array}$$

Note that these will be represented as *type families* since judgments are represented as types and deductions as objects.

$$\begin{array}{lcl} \mathsf{value} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathsf{exp}(t) \to \mathrm{type} \\ \mathsf{eval} & : & \forall t{:}\mathsf{tp}.\ \mathsf{exp}(t) \to \mathsf{exp}(t) \to \mathrm{type} \end{array}$$

Note that value and eval are both type families with dependent kinds, that is, the type of the second and third index objects depends on the first index object t in both cases. In the fragment we consider here, the evaluation rules are straightforward, although we have to take care to consume extraneous resources in the case of variable.

$$\begin{array}{lcl} \mathsf{ev\_true} & : & \top \multimap \mathsf{eval\ bool\ true\ true} \\ \mathsf{ev\_false} & : & \top \multimap \mathsf{eval\ bool\ false\ false} \\ \mathsf{ev\_var} & : & \forall t{:}\mathsf{tp}.\ \forall x{:}\mathsf{var}(t).\ \forall v{:}\mathsf{exp}(t).\ \forall e{:}\mathsf{exp}(t). \\ & & \top \multimap \mathsf{value}\,t\,x\,v \multimap \mathsf{eval}\,t\,(\mathsf{v}\,t\,x)\,v \end{array}$$

What did we need so far? We have used unrestricted implication and universal quantification, linear implication and additive truth. If we had a binary operator for expressions we would also need an additive conjunction so that the values of variables are accessible in both branches. Kinds and signatures change only to the extent that the types embedded in them change.

$$\begin{array}{rrcl} \text{Kinds} & K & ::= & \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ \mathrm{type} \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid c{:}A \\ \text{Types} & A & ::= & a\,M_1 \ldots M_n \mid \Pi x{:}A_1 \ldots A_2 \\ & & & \mid A_1 \multimap A_2 \mid A_1 \& A_2 \mid \top \\ \text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\,M_2 \\ & & & \mid \hat{\lambda} x{:}A.\ M \mid M_1\,\hat{}\,M_2 \\ & & & \mid \langle M_1, M_2 \rangle \mid \mathsf{fst}\,M \mid \mathsf{snd}\,M \\ & & & \mid \langle\,\rangle \end{array}$$

Note that in this fragment, uniform deductions are complete. Moreover, every term has a canonical form, which is a β-normal, η-long form. Canonical forms are defined as in Section 2.6 where the coercion from atomic to normal derivations is restricted to atomic propositions. This is important, because it allows a relatively simple algorithm for testing equality between objects, which is necessary because of the rule of type conversion in the type theory.

The appropriate notion of definitional equality here is then defined by the rules for β-reduction and η-expansion.

$$\frac{\Gamma, x{:}A; \Delta \vdash M : B \qquad \Gamma; \cdot \vdash N : A}{\Gamma; \Delta \vdash (\lambda x{:}A.\ M)\, N = [N/x]M : [N/x]B}$$

$$\frac{\Gamma; \Delta, u{:}A \vdash M : B \qquad \Gamma; \Delta' \vdash N : A}{\Gamma; \Delta, \Delta' \vdash (\hat{\lambda} u{:}A.\ M)\,\hat{}\, N = [N/u]M : B}$$

$$\frac{\Gamma; \Delta \vdash M_1 : A \qquad \Gamma; \Delta \vdash M_2 : B}{\Gamma; \Delta \vdash \mathsf{fst}\,\langle M_1, M_2\rangle = M_1 : A} \qquad \frac{\Gamma; \Delta \vdash M_1 : A \qquad \Gamma; \Delta \vdash M_2 : B}{\Gamma; \Delta \vdash \mathsf{snd}\,\langle M_1, M_2\rangle = M_2 : B}$$

$$\frac{\Gamma; \Delta \vdash M : \forall x{:}A.\ B}{\Gamma; \Delta \vdash M = \lambda x{:}A.\ M\, x : \forall x{:}A.\ B}$$

$$\frac{\Gamma; \Delta \vdash M : A \multimap B}{\Gamma; \Delta \vdash M = \hat{\lambda} u{:}A.\ M\,\hat{}\,u : A \multimap B}$$

$$\frac{\Gamma; \Delta \vdash M : A \& B}{\Gamma; \Delta \vdash M = \langle \mathsf{fst}\, M, \mathsf{snd}\, M\rangle : A \& B}$$

$$\frac{\Gamma; \Delta \vdash M : \top}{\Gamma; \Delta \vdash M = \langle\,\rangle : \top}$$

The remaining rules are reflexivity, symmetry, transitivity, and congruence rules for each constructors. It is by no means trivial that this kind of equality is decidable (see [CP98, VC00]). We would like to emphasize once again that in a dependent type theory, decidability of judgmental (definitional) equality is necessary to obtain a decidable type-checking problem.

Next we come to the specification of the operational semantics of commands. For this we give a natural specification in terms of two type families, $\mathsf{exec}\,D\,C$ where D is a label for the command, and $\mathsf{done}\,D$ which indicates that the command labeled D has finished. The labels here are not targets for jumps since they are not part of the program (see Exercise **??**). Note that there are no constructors for labels—we will generate them as parameters during the execution.

$$\begin{array}{rcl} \mathsf{lab} & : & \mathrm{type} \\ \mathsf{exec} & : & \mathsf{lab} \to \mathsf{cmd} \to \mathrm{type} \\ \mathsf{done} & : & \mathsf{lab} \to \mathrm{type} \end{array}$$

According to our general strategy, $\mathsf{exec}\ D\ C$ and $\mathsf{done}\ D$ will be part of our linear hypothesis. Without giving a structural operational semantics, we present the rules for execution of commands directly in the type theory.

skip returns immediately.

$$\mathsf{ex\_skip} \quad : \quad \forall D{:}\mathsf{lab}.\ \mathsf{exec}\ D\ \mathsf{skip} \multimap \mathsf{done}\ D$$

For sequential composition, we prevent execution of the second command until the first has finished.

$$\begin{array}{lcl} \mathsf{ex\_seq} & : & \forall D{:}\mathsf{lab}.\ \forall C_1{:}\mathsf{cmd}.\ \forall C_2{:}\mathsf{cmd}. \\ & & \quad \mathsf{exec}\ D\ (\mathsf{seq}\ C_1\ C_2) \\ & & \quad \multimap (\exists d_1{:}\mathsf{lab}.\ \mathsf{exec}\ d_1\ C_1 \otimes (\mathsf{done}\ d_1 \multimap \mathsf{exec}\ D\ C_2)) \end{array}$$

For parallel composition, both commands can proceed independently. The parallel composition is done, if both commands are finished. This is by no means the only choice of an operational semantics.

$$\begin{array}{lcl} \mathsf{ex\_par} & : & \forall D{:}\mathsf{lab}.\ \forall C_1{:}\mathsf{cmd}.\ \forall C_2{:}\mathsf{cmd}. \\ & & \quad \mathsf{exec}\ D\ (\mathsf{par}\ C_1\ C_2) \\ & & \quad \multimap \exists d_1{:}\mathsf{lab}.\ \exists d_2{:}\mathsf{lab}.\ \ \mathsf{exec}\ d_1\ C_1 \otimes \mathsf{exec}\ d_2\ C_2 \\ & & \qquad\qquad\qquad\qquad\qquad \otimes (\mathsf{done}\ d_1 \otimes \mathsf{done}\ d_2 \multimap \mathsf{done}\ D) \end{array}$$

For assignment, we need to consume the assumption $x = v'$ for the variable and assume the new value as $x = v$. We also want to allow assignment to an uninitialized variables, which is noted by an assumption $\mathsf{uninit}\ulcorner\tau\urcorner x$.

$$\begin{array}{lcl} \mathsf{uninit} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathrm{type} \\ \mathsf{ex\_assign} & : & \forall D{:}\mathsf{lab}.\ \forall T{:}\mathsf{tp}.\ \forall X{:}\mathsf{var}(T).\ \forall E{:}\mathsf{exp}(T).\ \forall V{:}\mathsf{exp}(T). \\ & & \quad \mathsf{exec}\ D\ (\mathsf{assign}\ T\ X\ E) \\ & & \quad \multimap\ (\mathsf{eval}\ T\ E\ V \multimap \mathbf{0}) \\ & & \qquad\quad \oplus\ ((\mathsf{uninit}\ T\ X \oplus \exists V'{:}\mathsf{exp}(T).\ \mathsf{value}\ T\ X\ V') \\ & & \qquad\qquad\quad \multimap \mathsf{value}\ T\ X\ V \otimes \mathsf{done}\ D) \end{array}$$

The new command creates a new, uninitialized variable.

$$\begin{array}{lcl} \mathsf{ex\_new} & : & \forall D{:}\mathsf{lab}.\ \forall T{:}\mathsf{tp}.\ \forall C{:}\mathsf{var}(T) \to \mathsf{cmd}. \\ & & \quad \mathsf{exec}\ D\ (\mathsf{new}\ T\ (\lambda x{:}\mathsf{var}(T).\ C(x))) \\ & & \quad \multimap \exists y{:}\mathsf{var}(T).\ \mathsf{uninit}\ T\ y \otimes \mathsf{exec}\ D\ C(y) \end{array}$$

Note type of C, which depends on a variable x. In order to emphasize this point, we have created a variable y with a different name on the right-hand side. The $\mathsf{ex\_new}$ constant will be applied to a (framework) function $(\lambda z{:}\mathsf{var}(T).\ C')$ which is applied to x on the left-hand side and y on the right hand side. We then have

$$C(x) = (\lambda z{:}\mathsf{var}(T).\ C')\,x = [x/z]C'$$

where the second equality is a *definitionaly equality* which follows by β-conversion. On the right-hand side have instead

$$C(y) = (\lambda z{:}\mathsf{var}(T).\ C')\,y = [y/z]C',$$

again by β-conversion.

So here we take critical advantage of the rule of type conversion in order to construct our encoding.

We leave the straightforward rules for the conditional to Exercise **??**.

For loops, we give two alternative formulations. The first take advantage of definitional equality in order to perform substitution, thereby unrolling the loop.

$$\begin{array}{lll} \mathsf{ex\_loop} & : & \forall D{:}\mathsf{lab}.\, \forall C{:}\mathsf{cmd} \to \mathsf{cmd} \\ & & \quad \mathsf{exec}\ D\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))) \\ & & \quad \multimap \mathsf{exec}\ D\ (C\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l)))) \end{array}$$

Assume the command has form $\mathsf{loop}\, l.\ C'$. Then

$$\ulcorner \mathbf{loop}\, l.\ C' \urcorner = \mathsf{loop}\ (\lambda l{:}\mathsf{lab}.\ \ulcorner C' \urcorner).$$

Then the framework variable C will be instantiated with

$$C = \lambda l{:}\mathsf{cmd}.\ C'.$$

For the framework expression on the right-hand side we obtain

$$\begin{array}{l} (C\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l)))) \\ = (\lambda l{:}\mathsf{cmd}.\ C')\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))) \\ = [(\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l)))/l]C' \\ = [\ulcorner \mathbf{loop}\, l.\ C' \urcorner/l]\ulcorner C' \urcorner \\ = \ulcorner [\mathbf{loop}\, l.\ C'/l]C' \urcorner \end{array}$$

Here the last equation follows by *compositionality*: substitution commutes with representation. This is a consequence of the decision to represent object-language variables by meta-language variables and can lead to very concise encodings (just as in this case). It means we do not have to explicitly axiomatize substitution, but we can let the framework take care of it for us. Since formalizing substitution can be a significant effort, this is a major advantage of higher-order abstract syntax over other representation techniques.

We can also avoid explicit substitution, instead adding an linear hypothesis that captures how to jump back to the beginning of a loop more directly.

$$\begin{array}{lll} \mathsf{ex\_loop}' & : & \forall D{:}\mathsf{lab}.\, \forall C{:}\mathsf{cmd} \to \mathsf{cmd} \\ & & \quad \mathsf{exec}\ D\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))) \\ & & \quad \multimap \exists k{:}\mathsf{cmd}.\ \ (\forall d{:}\mathsf{lab}.\ \mathsf{exec}\ d\ k \multimap \mathsf{exec}\ d\ (C(k))) \\ & & \qquad\qquad\qquad \otimes\ \mathsf{exec}\ D\ (C(k))) \end{array}$$

This completes our specification of the operational semantics, which is at a very high level of abstraction. Unfortunately, the concurrency in the specification requires leaving the fragment we have discussed so far and including multiplicative conjunction and existential quantification, at least. As mentioned before, there is a translation back into the uniform fragment that preserves provability. This translation, however, does not preserve a natural notion of equality on proofs, or a natural semantics in terms of logic program execution. It is a subject of current research to determine how concurrency can be introduced into the linear logical framework LLF in a way that preserves the right notion of equality.

So far, it seems we have not used the η-conversion. Our representation function is a bijection between syntactic categories of the object language and canonical forms of the representation type. Together with β-reduction, we need η-expansion to transform and arbitrary object to its canonical form. Without it we would have "exotic objects" that do not represent any expression in the language we are trying to model.

Bibliography

[ABCJ94] D. Albrecht, F. Bäuerle, J. N. Crossley, and J. S. Jeavons. Curry-Howard terms for linear logic. In ??, editor, *Logic Colloquium '94*, pages ??–?? ??, 1994.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[ACS98] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchrounous π-calculus. *Theoretical Computer Science*, 195(2):291–423, 1998.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[AP91] J.-M. Andreoli and R. Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schröder-Heister, editor, *Proceedings of Workshop to Extensions of Logic Programming, Tübingen, 1989*, pages 1–30. Springer-Verlag LNAI 475, 1991.

[AS01] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *Submitted*, 2001. A previous version presented at LICS'00.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[Bib86] Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[Bie94] G. Bierman. On intuitionistic linear logic. Technical Report 346, University of Cambridge, Computer Laboratory, August 1994. Revised version of PhD thesis.

[BM01] David Basin and Seán Matthews. Logical frameworks. In Dov Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition, 2001. In preparation.

[BS92] G. Bellin and P. J. Scott. On the π-calculus and linear logic. Manuscript, 1992.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Cer95] Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M.I. Sessa, editors, *Proceedings of the Joint Conference on Declarative Programming (GULP-PRODE'95)*, pages 313–318, Marina di Vietri, Italy, September 1995. Palladio Press.

[Cer96] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

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

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

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

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[CP98] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 1998. To appear in a special issue with invited papers from LICS'96, E. Clarke, editor.

[Doš93] Kosta Došen. A historical introduction to substructural logics. In Peter Schroeder-Heister and Kosta Došen, editors, *Substructural Logics*, pages 1–30. Clarendon Press, Oxford, England, 1993.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Translated under the title *Investigations into Logical Deductions* in [Sza69].

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir93] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[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.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[Hof00a] Martin Hofmann. Linear types and non-size increasing polynomial time computation. *Theoretical Computer Science*, 2000. To appear. A previous version was presented at LICS'99.

[Hof00b] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, November 2000. To appear. A previous version was presented as ESOP'00.

[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.

[HP97] James Harland and David Pym. Resource-distribution via boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer-Verlag LNAI 1249.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[IP98] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[Kni89] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[Lin92] P. Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

[Mil92] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML94] Per Martin-Löf. Analytic and synthetic judgements in type theory. In Paolo Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

[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.

[MM76] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[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.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[Pfe01] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001. In press.

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

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rob71] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Company, 1993. Also in Bulletin of the European Association for Theoretical Computer Science, volume 41, pages 154–165.

[SHD93] Peter Schroeder-Heister and Kosta Došen, editors. *Substructural Logics.* Number 2 in Studies in Logic and Computation. Clarendon Press, Oxford, England, 1993.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Co., Amsterdam, 1969.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal Of Symbolic Logic*, 32:198–212, 1967.

[Tro92] A. S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Tro93] A. S. Troelstra. Natural deduction for intuitionistic linear logic. Prepublication Series for Mathematical Logic and Foundations ML-93-09, Institute for Language, Logic and Computation, University of Amsterdam, 1993.

[VC00] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear LF. Draft paper, September 2000.

[WW01] David Walker and Kevin Watkins. On linear types and regions. In *Proceedings of the International Conference on Functional Programming (ICFP'01)*. ACM Press, September 2001.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.