Lecture Notes on
Sax: An Imperative Intermediate Language

15-416/816: HOT Compilation
Frank Pfenning

Lecture 1
January 14, 2025

1 Introduction

One of the key aspects of writing a compiler is to design appropriate *intermediate languages* that represent stepping stones between the source and the target. Source, intermediate, and target languages will evolve throughout the semester because you will implement a sequence of compilers for languages of increasing complexity. The goal of the first lab is to familiarize you with the intermediate language; the goal of the second lab will be to translate into and out of it.

All (or perhaps almost all?) compilers rely on intermediate languages. In 15-411 Compiler Design common intermediate languages are static single-assignment (SSA) or LLVM. The TIL compiler [Tarditi et al., 2004] for Standard ML developed at CMU uses multiple intermediate languages and is based on the premise that they should be *typed*, all the way down to typed assembly language (TAL) [Morrisett et al., 2003].

Sax [DeYoung et al., 2020] is a relatively recent entry into this space. It is quite high level in the sense that it corresponds to a logical system based on a form of the sequent calculus as an instance of the Curry-Howard correspondence. And yet it has an intuitive imperative interpretation. Because of its relatively high-level nature, it is easy to define and use *substructural* variants and explore how to implement them efficiently and take advantage of the properties they guarantee.

Instead of presenting all of Sax at once, we start with a core that is *positive* and *linear*. In this context, *positive* means that the language is first-order, that is, it doesn't have any higher-order functions. And *linear* here means that no traditional garbage collection is needed because all memory cells can be deallocated when they are read.

2 Positive Types and Large Values

From the user perspective, values of positive types are those whose structure we can directly observe. These would be booleans, natural numbers, lists or trees with elements of observable type, etc. Conversely, values of negative type are those whose structure we **cannot** directly observe, but can interact with like functions or objects.

The theory of programming languages has developed a basic understanding of the building blocks of types. Correspondingly, proof theory has identified building blocks of propositions. We'll mostly stick with the programming language point of view, but we will sometimes correlate it with intuitionistic (linear) propositions.

Our basic judgment for typing large values is

$$V : A$$

expressing that large value V has type A. The values we observe do not contain any variables, so we do not need any context to specify their type. We call these values *large* because a single value contains all the information about the outcome of a computation.

Pairs. We write $A \otimes B$ for the positive types of pairs.

$$\frac{V : A \quad W : B}{(V, W) : A \otimes B}\ \otimes I$$

The notation is borrowed from linear logic, as is the naming of the rule I as an *introduction rules*. In intuitionistic logic, this would be one of two forms of conjunction. In functional languages with a call-by-value semantics (the ML family), these are written `A * B`, and this is actually the concrete syntax we adopt.

Unit. We write $\mathbf{1}$ for the unit type, inhabited only by the unit element.

$$\frac{}{(\,) : \mathbf{1}}\ \mathbf{1}I$$

In common languages this is often written as **unit**; we'll just write `1` in our concrete syntax.

Sums. We write $\Sigma_{\ell \in L}(\ell : A_\ell)$ for the disjoint sum of the types A_ℓ, where the *labels* ℓ are drawn from a finite index set $L \neq \{\,\}$. In linear logic, the binary version is written as $A \oplus B$, which is the same as $A \vee B$ in intuitionistic logic. We avoid empty sums for simplicity. Also, since they contain no values, their programming applications are limited. We generally use ℓ and k for labels.

$$\frac{(k \in L) \quad V : A_k}{k(V) : \Sigma_{\ell \in L}(\ell : A_\ell)}\ \oplus I \qquad \left[\ \frac{V : A}{\underline{\mathsf{inl}}(V) : A \oplus B}\ \oplus I_1 \qquad \frac{V : B}{\underline{\mathsf{inr}}(V) : A \oplus B}\ \oplus I_2\ \right]$$

We have shown the special case of the binary version on the right, where $A \oplus B \triangleq (\underline{\mathsf{inl}} : A) + (\underline{\mathsf{inr}} : B)$, They aren't properly part of the language.

With these value constructors we can already represent the finite types such as the booleans $(\underline{\mathsf{false}} : \mathbf{1}) + (\underline{\mathsf{true}} : \mathbf{1})$ In order to represent infinite types like lists or natural numbers, we also need recursion.

Equirecursive Types. Rather than adding a binding constructor $\mu\alpha.\, A(\alpha)$ for constructing recursive types, we represent them by top-level definitions, like

$$\mathbf{type}\ \mathsf{nat} = (\underline{\mathsf{zero}} : \mathbf{1}) \oplus (\underline{\mathsf{succ}} : \mathsf{nat})$$

From now on we adopt a notation for sums that is closer to our concrete syntax.

$$\mathbf{type}\ \mathsf{nat} = \oplus\{\underline{\mathsf{zero}} : \mathbf{1}, \underline{\mathsf{succ}} : \mathsf{nat}\}$$

What does it mean for these types to be *equirecursive*? It means that we treat this definition as an *equation* at the meta-level, rather than as an *isomorphism* that requires coercions. We will see in the next lecture that this simplicity comes at some cost, which I believe is well worth paying.

Because type definitions are equirecursive we have judgments such as

$$\underline{\mathsf{succ}}(\underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,))) : \mathsf{nat}$$

We also don't have a separate rule for type names such as nat because we can "silently" apply the rules for sums to derive the above judgment.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\,) : \mathbf{1}}\ \mathbf{1}I}{\underline{\mathsf{zero}}(\,) : \mathsf{nat}}\ \oplus I}{\underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,)) : \mathsf{nat}}\ \oplus I}{\underline{\mathsf{succ}}(\underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,))) : \mathsf{nat}}\ \oplus I$$

In the mathematical sense all positive types defined in this way are *inductive*. This requires a restriction: the right-hand side of a type definition should not be a type name itself, but start with a constructor ($\otimes, \mathbf{1}, \oplus$). This avoid definitions such as $\mathbf{type}\ t = t$.

Furthermore, we allow all type definitions to be mutually recursive, which can be a bit unpleasant if we use an explicit type constructor μ.

As another example, lists of natural numbers could be defined with

$$\mathbf{type}\ \mathsf{list} = \oplus\{\underline{\mathsf{nil}} : \mathbf{1}, \underline{\mathsf{cons}} : \mathsf{nat} \otimes \mathsf{list}\}$$

3 Memory and Small Values

At this point we know we'd like to represent large values, but we want to somehow represent them in memory so we can operate on them imperatively. A memory cell, however, cannot hodl a value of arbitrary size, so we need to break large values down in smaller pieces. We do this using an abstract semantic object $\mathsf{cell}\ a\ v$, representing a memory cell at address a holding small value v. A small value is either a pair of addresses, or the unit value, or a label together with an address. For example, the number two from the example above could be represented as

$$\begin{array}{lll} \mathsf{cell} & 0 & \underline{\mathsf{succ}}(1) \\ \mathsf{cell} & 1 & \underline{\mathsf{succ}}(3) \\ \mathsf{cell} & 3 & \underline{\mathsf{zero}}(4) \\ \mathsf{cell} & 4 & (\,) \end{array}$$

Here, we have used integers to represent addresses, which is actually typical for real memory. We also assumed that every memory cell has enough space to hold any small value. Typically, two machine words would be sufficient to hold a pair of addresses, tagged values, and also the unit value.

We may already be able to foresee some optimizations here, like not assigning an explicit cell to hold the unit value. However, it is important not to get carried away with such considerations at this stage, but to implement the straightforward version first. For one, it is supported by theory and the implementation should be simple and therefore more likely to be correct than a more complex but possibly optimized one. For another, it provides us with a baseline so we can later measure what optimizations actually buy us.

Another point: the result of the program that can be observed is its large value, not its precise layout. In the example above, we skipped the address 2 to make this point. So at the end of evaluation a small program needs to traverse memory and print out the large value. This program will depend on certain choices of memory layout, for example, whether the unit value is explicitly represented or not.

4 Typing Memory

How do we assign types to a memory cell? A cell

$$\mathsf{cell}\ c\ (a,b)$$

for example defines the small value at address c and references the small values at address a and b. We therefore type it as

$$a : A, b : B \vdash \mathsf{cell}\ c\ (a,b) :: (c : A \otimes B)$$

There are two departures from the typing of large values. For one, there are now antecedents that stand for the types of other memory cells, here a and b. The other is that we also provide an address for the succedent (here c) to track where the small value is stored.

Generally, we like to maintain types when programs execute—an almost universal principle to make sure the result of computation satisfies the original expectation with regards to its type. Therefore, a command *writing* to a memory cell should be typed just the way the resulting cell is. This gives use the following rules.

$$\frac{}{a : A, b : B \vdash \mathbf{write}\ c\ (a,b) :: (c : A \otimes B)}\ \otimes X \qquad \frac{}{\cdot \vdash \mathbf{write}\ c\ (\,)\ :: (c : \mathbf{1})}\ \mathbf{1}X$$

$$\frac{(k \in L)}{a : A_k \vdash \mathbf{write}\ c\ k(a) :: (c : \Sigma_{\ell \in L}(\ell : A_\ell))}\ \oplus X$$

We have given them the name X because the stand for logical axioms, if we ignore the commands.

$$\frac{}{A, B \vdash A \otimes B}\ \otimes X \qquad \frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}X \qquad \frac{}{A \vdash A \oplus B}\ \oplus X_1 \quad \frac{}{B \vdash A \oplus B}\ \oplus X_2$$

So the rules maintain their correspondence to logical rules. The typing judgment now has the form

$$\underbrace{a_1 : A_1, \ldots, a_n : A_n}_{\text{read from}} \vdash P :: \underbrace{(c : C)}_{\text{write to}}$$

where $a_1, \ldots, a_n$ and c stand for addresses and P is a command to be executed. We often think of P as a *process*, since (as we will see) the language has a natural parallel interpretation. If we want to emphasize the sequential meaning we say *command*.

In our dynamics we have two forms for semantic objects. $\mathsf{cell}\ c\ v$ represents that the cell with address c holds small value v. We also write $\mathsf{proc}\ P$ which means the process P is executing. The dynamics of the write operation is then almost trivially specified:

$$\mathsf{proc}\ (\mathbf{write}\ c\ v) \quad \longrightarrow \quad \mathsf{cell}\ c\ v$$

5 Reading from Memory

Writing small values to memory was easily interpreted by the way memory cells themselves had to be typed. What about reading? When reading a pair, for example, we bind two variables to addresses, one to the first component and one to the second component. The corresponding construct is

$$\mathbf{read}\ c\ (x,y) \Rightarrow P(x,y)$$

where we have made the dependence of P on x and y explicit.

We can see that if $c : A \otimes B$ holds a small value (a, b), then x should be bound to a and y should be bound to b while we execute P. We write this as

$$\mathsf{cell}\ c\ (a, b), \mathsf{proc}\ (\mathbf{read}\ c\ (x, y) \Rightarrow P(x, y)) \quad \longrightarrow \quad \mathsf{proc}\ P(a, b)$$

What does this imply for typing? When we read from c we use it, and then we get to use a and b (represented in the program by x and y) when we type P.

$$\frac{\Gamma, x : A, y : B \vdash P(x, y) :: \delta}{\Gamma, c : A \otimes B \vdash \mathbf{read}\ c\ (x, y) \Rightarrow P(x, y) :: \delta}\ \otimes L$$

Here, we have written δ for something like $d : D$, naming the address where P eventually will have to write to. We often refer to d as the *destination*.

The unit type works analogously, but does not bind any variables.

$$\mathsf{cell}\ c\ (\,), \mathsf{proc}\ (\mathbf{read}\ c\ (\,) \Rightarrow P) \quad \longrightarrow \quad \mathsf{proc}\ P$$

And from the typing perspective:

$$\frac{\Gamma \vdash P :: \delta}{\Gamma, c : \mathbf{1} \vdash \mathbf{read}\ c\ (\,) \Rightarrow P :: \delta}\ \mathbf{1}L$$

To read sums the process will have to account for all possible summands.

$$\frac{(\Gamma, x_\ell : A_\ell \vdash P_\ell(x_\ell) :: \delta) \quad (\forall \ell \in L)}{\Gamma, c : \Sigma_{\ell \in L}(\ell : A_\ell) \vdash \mathbf{read}\ c\ \{\ell(x_\ell) \Rightarrow P_\ell(x_\ell)\}_{\ell \in L} :: \delta}\ \oplus L$$

The $\oplus L$ rule has one premise for each label in L, so that during execution of the command the correct branch is always available.

$$\mathsf{cell}\ c\ k(a), \mathsf{proc}\ (\mathbf{read}\ c\ \{\ell(x_\ell) \Rightarrow P_\ell(x_\ell)\}_{\ell \in L}) \quad \longrightarrow \quad \mathsf{proc}\ P_k(a) \quad (k \in L)$$

There rules are call L for *left rules* because they consider the types of antecedents, that is, on the left-hand side of the turnstile. They are the left rules from Gentzen's sequent calculus [Gentzen, 1935], which may be familiar.

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes L \qquad \frac{\Gamma \vdash C}{\Gamma, \mathbf{1} \vdash C}\ \mathbf{1}L \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\ \oplus L$$

We observe that, from the logical perspective, we keep all the left rules from the sequent calculus and replace the right rules by axioms. That why we call this the *semi-axiomatic sequent calculus*, or Sax for short.

6 Cut and Identity

We are missing two important rules from the sequent calculus, cut and identity.

$$\frac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}\ \mathsf{cut} \qquad \frac{}{A \vdash A}\ \mathsf{id}$$

What do they mean here? It turns out that cut allocates a fresh cell of memory.

$$\frac{\Gamma \vdash P(x) :: (x : A) \quad \Delta, x : A \vdash Q(x) :: \delta}{\Gamma, \Delta \vdash \textbf{cut}\ (x : A)\ P(x)\ Q(x) :: \delta}\ \mathsf{cut}$$

We can read this as a parallel or a sequential construct, where the latter is a particular schedule for the parallel program. The process

$$\textbf{cut}\ (x : A)\ P(x)\ Q(x)$$

allocates a new cell a (referred to as a in the program) and spawn a process $P(a)$ whose obligation is to write to a. The process $Q(a)$ will read from a. The interpretation of our typing judgment doesn't leave much room here. Note that a is called a *future* [Halstead, 1985, Pruiksma and Pfenning, 2022] and a point of synchronization: if $Q(a)$ tries to read from a before $P(a)$ has written to a, $Q(a)$ will block until it is written.

The sequential interpretation executes $P(a)$ in its entirety (which is, until it writes to its destination a) and only then executes $Q(a)$. Then, $Q(a)$ never has to check if a has been written, since it is guaranteed to have been written.

From the perspective of the dynamics, it is just

$$\mathsf{proc}\ (\textbf{cut}\ (x : A)\ P(x)\ Q(x)) \quad \longrightarrow \quad \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) \quad (a \text{ fresh})$$

We see that in this formulation of the dynamics, more than one "proc" object can be in the state. The abstract rules allow any of them with applicable rule to transition. There are, of course, ways to enforce formally that proc $P(a)$ completes before proc $Q(a)$ starts, but if our key properties (like the standard progress and preservation) do not depend on this we can also leave it informal. In a way, an strategy that is compatible with the rules is permissible for an implementation.

This way of specifying the dynamics is called *substructural operational semantics* (SSOS) [Pfenning, 2004, Pfenning and Simmons, 2009, Simmons, 2012] and is based on *multiset rewriting* [Cervesato and Scedrov, 2009] because the collection of semantic objects that make up the state of the computation are interpreted as a multiset. More about that in the next lecture.

The identity rule just moves a small value from one cell to another, by reading from one cell and writing in one step.

$$\frac{}{b : A \vdash \textbf{id}\ a\ b :: (a : A)}\ \mathsf{id}$$

with the dynamics

$$\mathsf{cell}\ b\ v, \mathsf{proc}\ (\textbf{id}\ a\ b) \quad \longrightarrow \quad \mathsf{cell}\ a\ v$$

7 Procedure Definitions

Our language so far has only positive types, so how is it possible to define procedures? (We say *procedures* because they are executed for their effect on the store, not like functions that take an argument and return a result.) We didn't discuss this in lecture, but we used it in the examples we live-coded.

This is somewhat important, because we want to define a first-order language before we allow first-class functions, objects, etc. To define a procedure named p we write

$$\textbf{proc}\ p\ (x : A)\ (y_1 : B_1)\ \ldots\ (y_n : B_n) = P(x, y_1, \ldots, y_n)$$

Every procedure p takes a destination first (here $x : A$) and then additional arguments y_i. When executed, both x and all y_i will be bound to addresses. To achieve that, we have a new construct

$$\textbf{call}\ p\ a\ b_1\ \ldots\ b_n$$

where the number, positions, and types of the arguments must match those in the definition of the procedure.

As for type definitions, procedure definition can be arbitrarily mutually recursive. This is unlike the syntactically expressed form of (mutual) recursion in languages such as Standard ML and OCaml.

We have said that procedures aren't functions, but what are they? It turns out they are metavariables in the sense of *contextual modal type theory* [Nanevski et al., 2008]. Think about a sequent

$$y_1 : B_1, \ldots, y_n : B_n \vdash p :: (x : A)$$

for some metavariable p. What is the type of p? It depends both on all antecedents and the succedent. In contextual modal type theory we might write

$$p :: A[y_1 : B_1, \ldots, y_n : B_n]$$

but here the destination x also has to be named, so it might be written as

$$p :: (y_1 : B_1, \ldots, y_n : B_n \vdash x : A)$$

So p stands for the derivation of a whole sequent, rather than an individual function such as $B \to A$. Moreover, p will be closed in the sense that all of its variables will be x and y_i, but nothing else. It is this property of being closed that allows us to arbitrarily reuse these procedure definition even in the type system is otherwise linear.

Will address the typing of procedure definitions and calls in the next lecture.

8 Some Sample Code

There are some choices to be made in the concrete syntax. We write $\oplus$ simply as `+`, and $\otimes$ as `*`. Also, all labels have the form `'<ident>`. Comments have the form `// ...\n` or `/* ... */`, properly nested in the latter case. Also, the branches of a `read` are enclosed in braces `{ ... }` and each alternative is preceded by a vertical bar `|`. The only exception is if there is just a single branch. More details can be found in the specification of Lab 1, which we live-coded.

We use indentation to offset the command writing to a freshly allocated cell from the reader, although the syntax makes is unambiguous. The file below is lec01.sax.

```
/* Lecture 1 */
/* 15-{4,6,8}17 Spring 2025 */

type nat = +{'zero : 1, 'succ : nat}

type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}

// two = 'b0 'b1 'e ()

proc zero (d : bin) =
  cut u : 1
```

```
    write u ()
  write d 'e(u)

proc one (d : bin) =
  cut x : bin
    call zero x
  write d 'b1(x)

proc succ (d : bin) (x : bin) =
  read x {
  | 'b0(x0) => write d 'b1(x0)
  | 'b1(x1) => cut y : bin
                 call succ y x1
               write d 'b0(y)
  | 'e(u) => read u ()
             call one d
  }

proc two (d : bin) =
  cut x : bin
    call one x
  call succ d x
```

Parameterless procedures (those with only a destination) can be executed, which here are `zero`, `one`, and `two`. They result in a final state from which the representation of the corresponding large values can be read off. The corresponding file, lec01.sax.val, written by the compiler, has the following contents.

```
value zero = 'e ()
value one = 'b1 'e ()
value two = 'b0 'b1 'e ()
```

9 Summary

We haven't discussed many properties of the system, including what "linearity" of the commands entails. We will return to this in the next lecture. Here is the summary of today's rules.

9.1 Typing

Assume a signature Σ with declarations $\overline{y : B} \vdash p :: (x : A)$. The definition of p is not relevant to type-checking procedures, but we have to check each procedure definition against its type (assuming all other definitions satisfies their types).

$$\frac{}{y : A \vdash \mathbf{id}\ x\ y :: (x : A)}\ \mathsf{id} \qquad \frac{\Gamma \vdash P :: (x : A) \quad \Delta, x : A \vdash Q :: \delta}{\Gamma, \Delta \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta}\ \mathsf{cut}$$

$$\frac{}{\cdot \vdash \mathbf{write}\ x\ (\,) :: (x : \mathbf{1})}\ \mathbf{1}X \qquad \frac{\Gamma \vdash P :: \delta}{\Gamma, x : \mathbf{1} \vdash \mathbf{read}\ x\ (\,) \Rightarrow Q :: \delta}\ \mathbf{1}L$$

$$\frac{}{x : A, y : B \vdash \mathbf{write}\ z\ (x, y) :: z : A \otimes B}\ {\otimes}X \qquad \frac{\Gamma, x : A, y : B \vdash Q :: \delta}{\Gamma, z : A \otimes B \vdash \mathbf{read}\ z\ (x, y) \Rightarrow Q :: \delta}\ {\otimes}R$$

$$\frac{(k \in L)}{y : A_k \vdash \mathbf{write}\ x\ k(y) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ {\oplus}X \qquad \frac{\Gamma, y : A_\ell \vdash Q_\ell :: \delta \quad (\forall \ell \in L)}{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ x\ \{\ell(y_\ell) \Rightarrow Q_\ell\}_{\ell \in L} :: \delta}\ {\oplus}L$$

$$\frac{(\overline{y : B} \vdash p :: (x : A)) \in \Sigma \quad \Gamma \vdash (\overline{b/y}) :: (\overline{y : B})}{\Gamma \vdash \mathbf{call}\ p\ a\ \overline{b} :: (a : A)}\ \mathsf{call}$$

$$\frac{\Gamma_1 \vdash \eta_1 :: \Delta_1 \quad \Gamma_2 \vdash \eta_2 :: \Delta_2}{\Gamma_1, \Gamma_2 \vdash (\eta_1, \eta_2) :: (\Delta_1, \Delta_2)} \qquad \frac{}{\cdot \vdash (\cdot) :: (\cdot)} \qquad \frac{}{y : A \vdash (y/x) :: (x : A)}$$

9.2 Dynamics

We assume an environment Σ with definition **proc** $p\ x\ \overline{y} = Q(x, \overline{y})$. The rules are multiset rewriting rules. They can be applied sequentially by always evaluating $P(a)$ fully (that is, until it writes to a) before $Q(a)$ in a cut.

The rules below represent a minor refactoring of the rules in lecture. Instead considering eavh kind of small value v separately, we have general rules and a way to pass a small value to a pattern matching construction.

$$\begin{array}{rlcll}
 & \mathsf{proc}\ (\mathbf{write}\ c\ v) & \longrightarrow & \mathsf{cell}\ c\ v & \\
\mathsf{cell}\ c\ v, & \mathsf{proc}\ (\mathbf{read}\ c\ K) & \longrightarrow & \mathsf{proc}\ (v \triangleright K) & \\
 & \mathsf{proc}\ (\mathbf{cut}\ (x : A)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a\ \text{fresh}) \\
\mathsf{cell}\ b\ v, & \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}\ a\ v & \\
 & \mathsf{proc}\ (\mathbf{call}\ p\ a\ \overline{b}) & \longrightarrow & \mathsf{proc}\ Q(a, \overline{b}) & (\mathbf{proc}\ p\ x\ \overline{y} = Q(x, \overline{y}) \in \Sigma
\end{array}$$

$$\begin{array}{rclcl}
(\,) & \triangleright & (\,) \Rightarrow Q & = & Q \\
(a, b) & \triangleright & (x, y) \Rightarrow Q(x, y) & = & Q(a, b) \\
k(a) & \triangleright & \{\ell(y_\ell) \Rightarrow Q_\ell(y_\ell)\}_{\ell \in L} & = & Q_k(a)
\end{array}$$

References

Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.

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.

Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 13(5):957–959, 2003.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008.

Frank Pfenning. Substructural operational semantics and linear destination-passing style. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302. Abstract of invited talk.

Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*, pages 101–110, Los Angeles, California, August 2009. IEEE Computer Society Press.

Klaas Pruiksma and Frank Pfenning. Back to futures. *Journal of Functional Programming*, 32:e6, 2022.

Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices*, 39(4):554–567, 2004.