CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Symbols; Mutable State

April 15, 2025

Symbols vs. Variables

A *symbol* is an atomic datum with no internal structure.

Whereas a variable is given meaning by substitution, a symbol is given meaning by a family of operations indexed by symbols. A symbol is just a name, or index, for a family of operations.

Many different interpretations may be given to symbols

- **fluid binding, dynamic classification, mutable storage, communication channels.**

A type is associated to each symbol whose interpretation depends on the particular application.

- **For example, in the case of mutable storage, the type of a symbol constrains the contents of the cell named by that symbol to values of that type.**

SPCF: PCF with Symbol Declaration

Declaring **new symbols for use within a specified *scope*.**

- **The expression new a ∼ ρ in e introduces a "new" symbol a with associated type ρ for use within e.**

- **The declared symbol a is "new" in that it is bound by the declaration within e and so may be renamed at will to ensure that it differs from any finite set of active symbols.**

- **Whereas the statics determines the scope of a declared symbol, its range of significance, or *extent*, is determined by the dynamics.**
  - **Scoped dynamics: a symbol can only be used within its scope**
  - **Scope-free dynamics: a symbol can exceed its scope**

SPCF: PCF with Symbol Declaration

The syntax for symbol declaration in **SPCF** is given by the following grammar:

$$\text{Exp} \quad e \quad ::= \quad \texttt{new}\{\tau\}(a.e) \quad \texttt{new}\, a \sim \tau\, \texttt{in}\, e \quad \text{generation}$$

The statics of symbol declaration makes use of a *signature*, or *symbol context*, that associates a type to each of a finite set of symbols. We use the letter Σ to range over signatures, which are finite sets of pairs $a \sim \tau$, where a is a symbol and τ is a type. The typing judgment $\Gamma \vdash_\Sigma e : \tau$ is parameterized by a signature Σ associating types to symbols. In effect, there is an infinite family of typing judgments, one for each choice of Σ. The expression $\texttt{new}\, a \sim \tau\, \texttt{in}\, e$ shifts from one instance of the family to another by adding a new symbol to Σ.

SPCF Statics

The statics of symbol declaration itself is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma, a \sim \rho} e : \tau \quad \tau \text{ mobile}}{\Gamma \vdash_{\Sigma} \mathtt{new}\{\rho\}(a.e) : \tau} \tag{31.1}$$

- **In a scoped dynamics, mobility is defined so that the computed value of a mobile type cannot depend on any symbol. By constraining the scope of a declaration to have mobile type, we can, under this interpretation, ensure that the extent of a symbol is confined to its scope.**
- **In a scope-free dynamics, every type is deemed mobile, because the dynamics ensures that the scope of a symbol is widened to accommodate the possibility that the value returned from the scope of a declaration may depend on the declared symbol.**
- **The term "mobile" reflects the informal idea that symbols may or may not be "moved" from the scope of their declaration according to the dynamics given to them. A scope-free dynamics allows symbols to be moved freely, whereas a scoped dynamics limits their range of motion.**

SPCF Scoped Dynamics

The scoped dynamics of symbol declaration is given by a transition judgment of the form $e \underset{\Sigma}{\mapsto} e'$ indexed by a signature Σ specifying the active symbols of the transition. Either e or e' may involve the symbols declared in Σ, but no others.

$$\frac{e \underset{\Sigma, a \sim \rho}{\longmapsto} e'}{\mathtt{new}\{\rho\}(a.e) \underset{\Sigma}{\mapsto} \mathtt{new}\{\rho\}(a.e')} \tag{31.2a}$$

$$\frac{e \ \mathsf{val}_\Sigma}{\mathtt{new}\{\rho\}(a.e) \underset{\Sigma}{\mapsto} e} \tag{31.2b}$$

The definition of the judgment τ mobile must be chosen to ensure that the following *mobility condition* is satisfied:

If τ *mobile,* $\vdash_{\Sigma, a \sim \rho} e : \tau$, *and* e $\mathsf{val}_{\Sigma, a \sim \rho}$, *then* $\vdash_\Sigma e : \tau$ *and* e val_Σ.

SPCF Scoped Dynamics

Theorem 31.1 (Preservation). *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$, *then* $\vdash_\Sigma e' : \tau$.

Proof By induction on the dynamics of symbol declaration. Rule (31.2a) follows by induction, applying rule (31.1). Rule (31.2b) follows from the condition on mobility. □

Theorem 31.2 (Progress). *If* $\vdash_\Sigma e : \tau$, *then either* $e \underset{\Sigma}{\mapsto} e'$, *or* e val_Σ.

Proof There is only one rule to consider, rule (31.1). By induction, we have either $e \underset{\Sigma, a \sim \rho}{\longmapsto} e'$, in which case rule (31.2a) applies, or e $\mathsf{val}_{\Sigma, a \sim \rho}$, in which case by the mobility condition we have e val_Σ, and hence rule (31.2b) applies. □

SPCF Scope-Free Dynamics

The scope-free dynamics of symbols is defined by a transition system between states of the form $\nu\,\Sigma\,\{\,e\,\}$, where Σ is a signature and e is an expression over this signature. The judgment $\nu\,\Sigma\,\{\,e\,\} \longmapsto \nu\,\Sigma'\,\{\,e'\,\}$ states that evaluation of e relative to symbols Σ results in the expression e' in the extension Σ' of Σ.

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{new}\{\rho\}(a.e)\,\} \longmapsto \nu\,\Sigma, a \sim \rho\,\{\,e\,\}} \tag{31.3}$$

Rule (31.3) specifies that symbol generation enriches the signature with the newly introduced symbol by extending the signature for all future transitions.

All other rules of the dynamics are changed to account for the allocated symbols. For example, the dynamics of function application cannot be inherited from Chapter 19 but is reformulated as follows:

$$\frac{\nu\,\Sigma\,\{\,e_1\,\} \longmapsto \nu\,\Sigma'\,\{\,e_1'\,\}}{\nu\,\Sigma\,\{\,e_1(e_2)\,\} \longmapsto \nu\,\Sigma'\,\{\,e_1'(e_2)\,\}} \tag{31.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\lambda\,(x:\tau)\,e(e_2)\,\} \longmapsto \nu\,\Sigma\,\{\,[e_2/x]e\,\}} \tag{31.4b}$$

SPCF Scope-Free Dynamics

Theorem 31.3 (Preservation)**.** *If* $\nu\ \Sigma\ \{e\} \longmapsto \nu\ \Sigma'\ \{e'\}$ *and* $\vdash_\Sigma e : \tau$*, then* $\Sigma' \supseteq \Sigma$ *and* $\vdash_{\Sigma'} e' : \tau$*.*

Proof There is only one rule to consider, rule (31.3), which is handled by inversion of rule (31.1). □

Theorem 31.4 (Progress)**.** *If* $\vdash_\Sigma e : \tau$*, then either* $e\ \mathsf{val}_\Sigma$ *or* $\nu\ \Sigma\ \{e\} \longmapsto \nu\ \Sigma'\ \{e'\}$ *for some* Σ' *and* e'*.*

Proof Immediate, by rule (31.3). □

PCF with Symbol References

A *symbol reference is* an expression whose purpose is to refer to a particular symbol.

Symbol references are values of a type ρ sym and are written 'a for some symbol a with associated type ρ.

The elimination form for the type ρ sym is a conditional branch that determines whether a symbol reference refers to a statically specified symbol.

The statics of the elimination form ensures that, in the positive case, the type associated to the referenced symbol is manifested, whereas in the negative case, no type information can be gleaned from the test.

PCF with Symbol References

Symbols are not themselves values, but they may be used to form values. One useful example is provided by the type τ `sym` of *symbol references*. A value of this type has the form $'a$, where a is a symbol in the signature. To compute with a reference, we may branch according to whether it is a reference to a specified symbol. The syntax of symbol references is given by the following grammar:

Typ	τ	::=	$\texttt{sym}(\tau)$	$\tau\,\texttt{sym}$	symbols
Exp	e		$\texttt{quote}[a]$	$'a$	reference
			$\texttt{is}[a]\{t.\tau\}(e;e_1;e_2)$	$\texttt{if}\,e\,\texttt{is}\,a\,\texttt{then}\,e_1\,\texttt{ow}\,e_2$	comparison

The expression $\texttt{quote}[a]$ is a reference to the symbol a, a value of type $\texttt{sym}(\tau)$. The expression $\texttt{is}[a]\{t.\tau\}(e;e_1;e_2)$ compares the value of e, which is a reference to some symbol b, with the given symbol a. If b is a, the expression evaluates to e_1, and otherwise to e_2.

PCF with Symbol References

The typing rules for symbol references are as follows:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \rho} \mathtt{quote}[a] : \mathtt{sym}(\rho)} \tag{31.5a}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \rho} e : \mathtt{sym}(\rho') \quad \Gamma \vdash_{\Sigma, a \sim \rho} e_1 : [\rho/t]\tau \quad \Gamma \vdash_{\Sigma, a \sim \rho} e_2 : [\rho'/t]\tau}{\Gamma \vdash_{\Sigma, a \sim \rho} \mathtt{is}[a]\{t.\tau\}(e; e_1; e_2) : [\rho'/t]\tau} \tag{31.5b}$$

PCF with Symbol References

The (scoped) dynamics of symbol references is given by the following rules:

$$\frac{}{\mathtt{quote}[a]\ \mathsf{val}_{\Sigma,a\sim\rho}} \tag{31.6a}$$

$$\frac{}{\mathtt{is}[a]\{t.\tau\}(\mathtt{quote}[a];e_1;e_2) \underset{\Sigma,a\sim\rho}{\longmapsto} e_1} \tag{31.6b}$$

$$\frac{(a \neq a')}{\mathtt{is}[a]\{t.\tau\}(\mathtt{quote}[a'];e_1;e_2) \underset{\Sigma,a\sim\rho,a'\sim\rho'}{\longmapsto} e_2} \tag{31.6c}$$

$$\frac{e \underset{\Sigma,a\sim\rho}{\longmapsto} e'}{\mathtt{is}[a]\{t.\tau\}(e;e_1;e_2) \underset{\Sigma,a\sim\rho}{\longmapsto} \mathtt{is}[a]\{t.\tau\}(e';e_1;e_2)} \tag{31.6d}$$

Rules (31.6b) and (31.6c) specify that $\mathtt{is}[a]\{t.\tau\}(e;e_1;e_2)$ branches according to whether the value of e is a reference to the symbol a.

PCF with Symbol References

Theorem 31.5 (Preservation)**.** *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$*, then* $\vdash_\Sigma e' : \tau$*.*

Proof By rule induction on rules (31.6). The most interesting case is rule (31.6b). When the comparison is positive, the types ρ and ρ' must be the same, because each symbol has at most one associated type. Therefore, e_1, which has type $[\rho'/t]\tau$, also has type $[\rho/t]\tau$, as required. □

Lemma 31.6 (Canonical Forms)**.** *If* $\vdash_\Sigma e : \mathtt{sym}(\rho)$ *and* $e\ \mathsf{val}_\Sigma$*, then* $e = \mathtt{quote}[a]$ *for some* a *such that* $\Sigma = \Sigma'$, $a \sim \rho$*.*

Proof By rule induction on rules (31.5), taking account of the definition of values. □

Theorem 31.7 (Progress)**.** *Suppose that* $\vdash_\Sigma e : \tau$*. Then either* $e\ \mathsf{val}_\Sigma$*, or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$*.*

Proof By rule induction on rules (31.5). For example, consider rule (31.5b), in which we have that $\mathtt{is}[a]\{t.\tau\}(e; e_1; e_2)$ has some type τ and that $e : \mathtt{sym}(\rho)$ for some ρ. By induction either rule (31.6d) applies, or else we have that $e\ \mathsf{val}_\Sigma$, in which case we are assured by Lemma 31.6 that e is $\mathtt{quote}[a]$ for some symbol b of type ρ declared in Σ. But then progress is assured by rules (31.6b) and (31.6c), because equality of symbols is decidable (either a is b or it is not). □

FSPCF: SPCF with Fluid Binding

Fluid binding associates to a symbol (and not a variable) a value of a specified type within a specified scope. The identification principle for bound variables is retained, type safety is not compromised, yet some of the benefits of dynamic scoping are preserved.

To account for fluid binding, we enrich **SPCF** defined in Chapter 31 with these constructs to obtain **FSPCF**:

Exp	e	::=	$\texttt{put}[a](e_1; e_2)$	$\texttt{put}\, e_1\, \texttt{for}\, a\, \texttt{in}\, e_2$	binding
			$\texttt{get}[a]$	$\texttt{get}\, a$	retrieval

The expression $\texttt{get}[a]$ evaluates to the value of the current binding of a, if it has one, and is stuck otherwise. The expression $\texttt{put}[a](e_1; e_2)$ binds the symbol a to the value e_1 for the duration of the evaluation of e_2, at which point the binding of a reverts to what it was prior to the execution. The symbol a is not bound by the put expression but is instead a parameter of it.

FSPCF Statics

The statics of **FSPCF** is defined by judgments of the form

$$\Gamma \vdash_\Sigma e : \tau,$$

much as in Chapter 31, except that here the signature associates a type to each symbol, instead of just declaring the symbol to be in scope. Thus, Σ is here defined to be a finite set of declarations of the form $a \sim \tau$ such that no symbol is declared more than once in the same signature. Note that the association of a type to a symbol is *not* a typing assumption. In particular, the signature Σ enjoys no structural properties and cannot be considered as a form of hypothesis as defined in Chapter 3.

The following rules govern the new expression forms:

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{get}[a] : \tau} \tag{32.1a}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau_1} e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma, a\sim\tau_1} e_2 : \tau_2}{\Gamma \vdash_{\Sigma, a\sim\tau_1} \mathtt{put}[a](e_1; e_2) : \tau_2} \tag{32.1b}$$

Rule (32.1b) specifies that the symbol a is a parameter of the expression that must be declared in Σ.

FSPCF Dynamics

The dynamics of **FSPCF** relies on a stack-like allocation of symbols in **SPCF** and maintains an association of values to symbols that tracks this stack-like allocation discipline. To do so, we define a family of transition judgments of the form $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$, where Σ is as in the statics, and μ is a finite function mapping some subset of the symbols declared in Σ to values of the right type. If μ is defined for some symbol a, then it has the form $\mu' \otimes a \hookrightarrow e$ for some μ' and value e. If μ is undefined for some symbol a, we may regard it as having the form $\mu' \otimes a \hookrightarrow \bullet$. We will write $a \hookrightarrow \_$ to stand for either $a \hookrightarrow \bullet$ or $a \hookrightarrow e$ for some expression e.

FSPCF Dynamics

The dynamics of **FSPCF** is defined by the following rules:

$$\frac{}{\mathtt{get}[a] \xmapsto[\Sigma, a \sim \tau]{\mu \otimes a \hookrightarrow e} e} \tag{32.2a}$$

$$\frac{e_1 \xmapsto[\Sigma, a \sim \tau]{\mu} e_1'}{\mathtt{put}[a](e_1; e_2) \xmapsto[\Sigma, a \sim \tau]{\mu} \mathtt{put}[a](e_1'; e_2)} \tag{32.2b}$$

$$\frac{e_1 \ \mathsf{val}_{\Sigma, a \sim \tau} \quad e_2 \xmapsto[\Sigma, a \sim \tau]{\mu \otimes a \hookrightarrow e_1} e_2'}{\mathtt{put}[a](e_1; e_2) \xmapsto[\Sigma, a \sim \tau]{\mu \otimes a \hookrightarrow -} \mathtt{put}[a](e_1; e_2')} \tag{32.2c}$$

$$\frac{e_1 \ \mathsf{val}_{\Sigma, a \sim \tau} \quad e_2 \ \mathsf{val}_{\Sigma, a \sim \tau}}{\mathtt{put}[a](e_1; e_2) \xmapsto[\Sigma, a \sim \tau]{\mu} e_2} \tag{32.2d}$$

FSPCF Dynamics

According to the dynamics of **FSPCF** given by rules (32.2), there is no transition of the form $\texttt{get}[a] \overset{\mu}{\underset{\Sigma}{\mapsto}} e$ if $\mu(a) = \bullet$. The judgment $e\ \mathsf{unbound}_\Sigma$ states that execution of e will lead to such a "stuck" state and is inductively defined by the following rules:

$$\frac{\mu(a) = \bullet}{\texttt{get}[a]\ \mathsf{unbound}_\mu} \tag{32.3a}$$

$$\frac{e_1\ \mathsf{unbound}_\mu}{\texttt{put}[a](e_1; e_2)\ \mathsf{unbound}_\mu} \tag{32.3b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{unbound}_\mu}{\texttt{put}[a](e_1; e_2)\ \mathsf{unbound}_\mu} \tag{32.3c}$$

In a larger language, it would also be necessary to include error propagation rules of the sort discussed in Chapter 6.

FSPCF Type Safety

We first define the auxiliary judgment $\mu : \Sigma$ by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{32.4a}$$

$$\frac{\vdash_\Sigma e : \tau \quad \mu : \Sigma}{\mu \otimes a \hookrightarrow e : \Sigma, a \sim \tau} \tag{32.4b}$$

$$\frac{\mu : \Sigma}{\mu \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau} \tag{32.4c}$$

These rules specify that, if a symbol is bound to a value, then that value must be of the type associated to the symbol by Σ. No demand is made in the case that the symbol is unbound (equivalently, bound to a "black hole").

Theorem 32.1 (Preservation)**.** *If* $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$*, where* $\mu : \Sigma$ *and* $\vdash_\Sigma e : \tau$*, then* $\vdash_\Sigma e' : \tau$*.*

Proof By rule induction on rules (32.2). Rule (32.2a) is handled by the definition of $\mu : \Sigma$. Rule (32.2b) follows by induction. Rule (32.2d) is handled by inversion of rules (32.1). Finally, rule (32.2c) is handled by inversion of rules (32.1) and induction. □

Theorem 32.2 (Progress)**.** *If* $\vdash_\Sigma e : \tau$ *and* $\mu : \Sigma$*, then either* $e\ \mathsf{val}_\Sigma$*, or* $e\ \mathsf{unbound}_\mu$*, or there exists* e' *such that* $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$*.*

FSPCF Subtleties

But what if the type of $\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2$ is a function type, so that the returned value is a λ-abstraction? The body of the returned λ may refer to the binding of a, which is reverted upon return from the `put`. For example, consider the expression

$$\texttt{put}\,17\,\texttt{for}\,a\,\texttt{in}\,\lambda\,(x:\texttt{nat})\,x + \texttt{get}\,a, \tag{32.5}$$

which has type $\texttt{nat} \rightharpoonup \texttt{nat}$, given that a is a symbol of type `nat`. Let us assume, for the sake of discussion, that a is unbound at the point at which this expression is evaluated. Evaluating the `put` binds a to the number 17 and returns the function $\lambda\,(x:\texttt{nat})\,x + \texttt{get}\,a$. But because a is reverted to its unbound state upon exiting the `put`, applying this function to an argument will result in an error, unless a binding for a is given. Thus, if f is bound to the result of evaluating (32.5), then the expression

$$\texttt{put}\,21\,\texttt{for}\,a\,\texttt{in}\,f(7) \tag{32.6}$$

will evaluate to 28, whereas evaluation of $f(7)$ in the absence of a surrounding binding for a will incur an error.

FSPCF Subtleties

One way to think about this situation is to consider that fluid-bound symbols serve as an alternative to passing extra arguments to a function to specialize its value when it is called. To see this, let e stand for the value of expression (32.5), a λ-abstraction whose body is dependent on the binding of the symbol a. To use this function safely, it is necessary that the programmer provide a binding for a prior to calling it. For example, the expression

$$\texttt{put}\ 7\ \texttt{for}\ a\ \texttt{in}\ (e(9))$$

evaluates to 16, and the expression

$$\texttt{put}\ 8\ \texttt{for}\ a\ \texttt{in}\ (e(9))$$

evaluates to 17. Writing just $e(9)$, without a surrounding binding for a, results in a run-time error attempting to retrieve the binding of the unbound symbol a.

This behavior can be simulated by adding an argument to the function value that will be bound to the current binding of the symbol a at the point where the function is called. Instead of using fluid binding, we would provide an extra argument at each call site, writing

$$e'(7)(9)$$

and

$$e'(8)(9),$$

respectively, where e' is the λ-abstraction

$$\lambda\,(y : \texttt{nat})\,\lambda\,(x : \texttt{nat})\,x + y.$$

FSPCF Subtleties

Adding arguments can be cumbersome, though, especially when several call sites provide the same binding for a. Using fluid binding, we may write

$$\texttt{put}\ 7\ \texttt{for}\ a\ \texttt{in}\ \langle e(8), e(9)\rangle,$$

whereas using an extra argument we must write

$$\langle e'(7)(8), e'(7)(9)\rangle.$$

However, such redundancy can be reduced by factoring out the common part, writing

$$\texttt{let}\ f\ \texttt{be}\ e'(7)\ \texttt{in}\ \langle f(8),\ f(9)\rangle.$$

FSPCF + Fluid References

We may extend **FSPCF** with fluid references by adding the following syntax:

Typ	τ	$::=$	$\texttt{fluid}(\tau)$	$\tau\ \texttt{fluid}$	fluid
Exp	e	$::=$	$\texttt{fl}[a]$	$\&\, a$	reference
			$\texttt{getfl}(e)$	$\texttt{getfl}\ e$	retrieval
			$\texttt{putfl}(e; e_1; e_2)$	$\texttt{putfl}\ e\ \texttt{is}\ e_1\ \texttt{in}\ e_2$	binding

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{fl}[a] : \texttt{fluid}(\tau)} \tag{32.8a}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{fluid}(\tau)}{\Gamma \vdash_{\Sigma} \texttt{getfl}(e) : \tau} \tag{32.8b}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{fluid}(\tau) \quad \Gamma \vdash_{\Sigma} e_1 : \tau \quad \Gamma \vdash_{\Sigma} e_2 : \tau_2}{\Gamma \vdash_{\Sigma} \texttt{putfl}(e; e_1; e_2) : \tau_2} \tag{32.8c}$$

FSPCF + Fluid References

The dynamics of references consists of resolving the referent and deferring to the underlying primitives acting on symbols.

$$\frac{}{\mathtt{fl}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}} \tag{32.9a}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'}{\mathtt{getfl}(e) \overset{\mu}{\underset{\Sigma}{\mapsto}} \mathtt{getfl}(e')} \tag{32.9b}$$

$$\frac{}{\mathtt{getfl}(\mathtt{fl}[a]) \overset{\mu}{\underset{\Sigma}{\mapsto}} \mathtt{get}[a]} \tag{32.9c}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'}{\mathtt{putfl}(e;e_1;e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} \mathtt{putfl}(e';e_1;e_2)} \tag{32.9d}$$

$$\frac{}{\mathtt{putfl}(\mathtt{fl}[a];e_1;e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} \mathtt{put}[a](e_1;e_2)} \tag{32.9e}$$

Modernized Algol (MA): PCF+Commands

Modernized Algol, or MA, is an imperative, block-structured programming language based on the classic language Algol.

MA extends PCF with a new syntactic sort of *commands* that act on *assignables* by retrieving and altering their contents. Assignables are introduced by *declaring* them for use within a specified scope; this is the essence of block structure. Commands are combined by sequencing and are iterated using recursion.

MA maintains a careful separation between *pure* expressions, whose meaning does not depend on any assignables, and *impure* commands, whose meaning is given in terms of assignables.

A distinctive feature of MA is that it adheres to the *stack discipline*, which means that assignables are allocated on entry to the scope of their declaration, and deallocated on exit, using a conventional stack discipline.

MA Syntax: PCF + Commands

The syntax of **MA** is given by the following grammar, from which we have omitted repetition of the expression syntax of **PCF** for the sake of brevity.

Typ	τ	$::=$	$\mathtt{cmd}$	$\mathtt{cmd}$	command
Exp	e	$::=$	$\mathtt{cmd}(m)$	$\mathtt{cmd}\, m$	encapsulation
Cmd	m	$::=$	$\mathtt{ret}(e)$	$\mathtt{ret}\, e$	return
			$\mathtt{bnd}(e; x.m)$	$\mathtt{bnd}\, x \leftarrow e\, ;\, m$	sequence
			$\mathtt{dcl}(e; a.m)$	$\mathtt{dcl}\, a := e\, \mathtt{in}\, m$	new assignable
			$\mathtt{get}[a]$	$@\, a$	fetch
			$\mathtt{set}[a](e)$	$a := e$	assign

The syntax of **PCF** is given by the following grammar:

Typ	τ	$::=$	$\mathtt{nat}$	$\mathtt{nat}$	naturals
			$\mathtt{parr}(\tau_1; \tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial function
Exp	e	$::=$	x	x	variable
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
			$\mathtt{ifz}\{e_0; x.e_1\}(e)$	$\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$	zero test
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\, (x : \tau)\, e$	abstraction
			$\mathtt{ap}(e_1; e_2)$	$e_1(e_2)$	application
			$\mathtt{fix}\{\tau\}(x.e)$	$\mathtt{fix}\, x : \tau\, \mathtt{is}\, e$	recursion

MA Statics

The statics of **MA** consists of two forms of judgment:

1. Expression typing: $\Gamma \vdash_\Sigma e : \tau$.
2. Command formation: $\Gamma \vdash_\Sigma m \ \mathsf{ok}$.

$$\frac{\Gamma \vdash_\Sigma m \ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}} \tag{34.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat}}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \ \mathsf{ok}} \tag{34.1b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd} \quad \Gamma, x : \mathtt{nat} \vdash_\Sigma m \ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x.m) \ \mathsf{ok}} \tag{34.1c}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat} \quad \Gamma \vdash_{\Sigma,a} m \ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a.m) \ \mathsf{ok}} \tag{34.1d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \mathtt{get}[a] \ \mathsf{ok}} \tag{34.1e}$$

$$\frac{\Gamma \vdash_{\Sigma,a} e : \mathtt{nat}}{\Gamma \vdash_{\Sigma,a} \mathtt{set}[a](e) \ \mathsf{ok}} \tag{34.1f}$$

MA Dynamics

The dynamics of **MA** is defined in terms of a *memory* μ a finite function assigning a numeral to each of a finite set of assignables.

The dynamics of expressions consists of these two judgment forms:

1. $e\ \mathsf{val}_\Sigma$, stating that e is a value relative to Σ.
2. $e \underset{\Sigma}{\mapsto} e'$, stating that the expression e steps to the expression e'.

$$\overline{\mathtt{cmd}(m)\ \mathsf{val}_\Sigma} \tag{34.2a}$$

Rule (34.2a) states that an encapsulated command is a value.

The dynamics of commands is defined in terms of states $m \parallel \mu$, where μ is a memory mapping assignables to values, and m is a command. There are two judgments governing such states:

1. $m \parallel \mu\ \mathsf{final}_\Sigma$. The state $m \parallel \mu$ is complete.
2. $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$. The state $m \parallel \mu$ steps to the state $m' \parallel \mu'$; the set of active assignables is given by the signature Σ.

MA Dynamics

These judgments are inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{ret}(e)\ \|\ \mu\ \mathsf{final}_\Sigma} \tag{34.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{ret}(e)\ \|\ \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e')\ \|\ \mu} \tag{34.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{bnd}(e; x.m)\ \|\ \mu \underset{\Sigma}{\mapsto} \mathtt{bnd}(e'; x.m)\ \|\ \mu} \tag{34.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e)); x.m)\ \|\ \mu \underset{\Sigma}{\mapsto} [e/x]m\ \|\ \mu} \tag{34.3d}$$

$$\frac{m_1\ \|\ \mu \underset{\Sigma}{\mapsto} m_1'\ \|\ \mu'}{\mathtt{bnd}(\mathtt{cmd}(m_1); x.m_2)\ \|\ \mu \underset{\Sigma}{\mapsto} \mathtt{bnd}(\mathtt{cmd}(m_1'); x.m_2)\ \|\ \mu'} \tag{34.3e}$$

MA Dynamics

$$\frac{}{\mathtt{get}[a] \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e} \tag{34.3f}$$

$$\frac{e \underset{\Sigma,a}{\longmapsto} e'}{\mathtt{set}[a](e) \parallel \mu \underset{\Sigma,a}{\longmapsto} \mathtt{set}[a](e') \parallel \mu} \tag{34.3g}$$

$$\frac{e \ \mathsf{val}_{\Sigma,a}}{\mathtt{set}[a](e) \parallel \mu \otimes a \hookrightarrow \_ \underset{\Sigma,a}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e} \tag{34.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{dcl}(e; a.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e'; a.m) \parallel \mu} \tag{34.3i}$$

$$\frac{e \ \mathsf{val}_{\Sigma} \quad m \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes a \hookrightarrow e'}{\mathtt{dcl}(e; a.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e'; a.m') \parallel \mu'} \tag{34.3j}$$

$$\frac{e \ \mathsf{val}_{\Sigma} \quad e' \ \mathsf{val}_{\Sigma,a}}{\mathtt{dcl}(e; a.\mathtt{ret}(e')) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e') \parallel \mu} \tag{34.3k}$$

MA Type Safety

The judgment $m \parallel \mu \ \mathsf{ok}_\Sigma$ is defined by the rule

$$\frac{\vdash_\Sigma m \ \mathsf{ok} \quad \mu : \Sigma}{m \parallel \mu \ \mathsf{ok}_\Sigma} \tag{34.4}$$

where the auxiliary judgment $\mu : \Sigma$ is defined by the rule

$$\frac{\forall a \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } e \ \mathsf{val}_\emptyset \text{ and } \vdash_\emptyset e : \mathtt{nat}}{\mu : \Sigma} \tag{34.5}$$

Theorem 34.1 (Preservation)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_\Sigma e : \tau$*, then* $\vdash_\Sigma e' : \tau$*.*
2. *If* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$*, with* $\vdash_\Sigma m \ \mathsf{ok}$ *and* $\mu : \Sigma$*, then* $\vdash_\Sigma m' \ \mathsf{ok}$ *and* $\mu' : \Sigma$*.*

Theorem 34.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* $e \ \mathsf{val}_\Sigma$*, or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$*.*
2. *If* $\vdash_\Sigma m \ \mathsf{ok}$ *and* $\mu : \Sigma$*, then either* $m \parallel \mu \ \mathsf{final}_\Sigma$ *or* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ *for some* μ' *and* m'*.*

MA Programming Idioms

We define the *sequential composition* of commands, written $\{x \leftarrow m_1 \,;\, m_2\}$, to stand for the command $\texttt{bnd}\, x \leftarrow \texttt{cmd}\,(m_1) \,;\, m_2$. Binary composition readily generalizes to an n-ary form by defining

$$\{x_1 \leftarrow m_1 \,;\, \ldots\, x_{n-1} \leftarrow m_{n-1} \,;\, m_n\},$$

to stand for the iterated composition

$$\{x_1 \leftarrow m_1 \,;\, \ldots\, \{x_{n-1} \leftarrow m_{n-1} \,;\, m_n\}\}.$$

We sometimes write just $\{m_1 \,;\, m_2\}$ for the composition $\{\_ \leftarrow m_1 \,;\, m_2\}$ where the returned value from m_1 is ignored; this generalizes in the obvious way to an n-ary form.

MA Programming Idioms

A related idiom, the command $\texttt{do}\, e$, executes an encapsulated command and returns its result. By definition, $\texttt{do}\, e$ stands for the command $\texttt{bnd}\, x \leftarrow e\, ;\, \texttt{ret}\, x$.

The *conditional* command $\texttt{if}\, (m)\, m_1\, \texttt{else}\, m_2$ executes either m_1 or m_2 according to whether the result of executing m is zero or not:

$$\{x \leftarrow m\, ;\, \texttt{do}\, (\texttt{ifz}\, x\, \{\texttt{z} \hookrightarrow \texttt{cmd}\, m_1 \mid \texttt{s}(\_) \hookrightarrow \texttt{cmd}\, m_2\})\}.$$

The returned value of the conditional is the value returned by the selected command.

The *while loop* command $\texttt{while}\, (m_1)\, m_2$ repeatedly executes the command m_2 while the command m_1 yields a non-zero number. It is defined as follows:

$$\texttt{do}\, (\texttt{fix}\, \mathit{loop} : \texttt{cmd}\, \texttt{is}\, \texttt{cmd}\, (\texttt{if}\, (m_1)\, \{\texttt{ret}\, \texttt{z}\}\, \texttt{else}\, \{m_2\, ;\, \texttt{do}\, \mathit{loop}\})).$$

A *procedure* is a function of type $\tau \rightharpoonup \texttt{cmd}$ that takes an argument of some type τ and yields an unexecuted command as result. Many procedures have the form $\lambda\, (x : \tau)\, \texttt{cmd}\, m$, which we abbreviate to $\texttt{proc}\, (x : \tau)\, m$. A *procedure call* is the composition of a function application with the activation of the resulting command. If e_1 is a procedure and e_2 is its argument, then the procedure call $\texttt{call}\, e_1(e_2)$ is defined to be the command $\texttt{do}\, (e_1(e_2))$, which immediately runs the result of applying e_1 to e_2.

MA Programming Idioms

As an example, here is a procedure of type `nat` $\rightharpoonup$ `cmd` that returns the factorial of its argument:

```
proc (x:nat) {
  dcl r := 1 in
  dcl a := x in
  { while ( @ a ) {
      y ← @ r
    ; z ← @ a
    ; r := (x-z+1)× y
    ; a := z-1
    }
    ; x ← @ r
    ; ret x
  }
}
```

MA + Typed Commands / Assignables

To admit declarations that return values other than `nat` and to admit assignables with contents of types other than `nat`, we must rework the statics of **MA** to record the returned type of a command and to record the type of the contents of each assignable. First, we generalize the finite set Σ of active assignables to assign a mobile type to each active assignable so that Σ has the form of a finite set of assumptions of the form $a \sim \tau$, where a is an assignable. Second, we replace the judgment $\Gamma \vdash_\Sigma m \ \mathsf{ok}$ by the more general form $\Gamma \vdash_\Sigma m \div \tau$, stating that m is a well-formed command returning a value of type τ. Third, the type `cmd` is generalized to $\mathtt{cmd}(\tau)$, which is written in examples as τ `cmd`, to specify the return type of the encapsulated command.

MA + Typed Commands / Assignables

The statics given in Section 34.1.1 is generalized to admit typed commands and typed assignables as follows:

$$\frac{\Gamma \vdash_\Sigma m \div \tau}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}(\tau)} \tag{34.6a}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \div \tau} \tag{34.6b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\tau) \quad \Gamma, x : \tau \vdash_\Sigma m \div \tau'}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x.m) \div \tau'} \tag{34.6c}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \text{ mobile} \quad \Gamma \vdash_{\Sigma, a \sim \tau} m \div \tau' \quad \tau' \text{ mobile}}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a.m) \div \tau'} \tag{34.6d}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] \div \tau} \tag{34.6e}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{set}[a](e) \div \tau} \tag{34.6f}$$

MA + Typed Commands / Assignables

Apart from the generalization to track returned types and content types, the most important change is that in rule (34.6d) both the type of a declared assignable and the return type of the declaration is required to be *mobile*. The definition of the judgment τ mobile is guided by the following *mobility condition*:

$$\textit{if } \tau \text{ mobile}, \vdash_\Sigma e : \tau \textit{ and } e \text{ val}_\Sigma, \textit{ then } \vdash_\emptyset e : \tau \textit{ and } e \text{ val}_\emptyset. \tag{34.7}$$

That is, a value of mobile type may not depend on any active assignables.

As long as the successor operation is evaluated eagerly, the type `nat` is mobile:

$$\frac{}{\texttt{nat mobile}} \tag{34.8}$$

Similarly, a product of mobile types may safely be deemed mobile, if pairs are evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 \times \tau_2 \text{ mobile}} \tag{34.9}$$

And the same goes for sums, if the injections are evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 + \tau_2 \text{ mobile}} \tag{34.10}$$

MA + Typed Commands / Assignables

The mobility restriction on the statics of declarations ensures that the type associated to an assignable is always mobile. We may therefore assume, without loss of generality, that the types associated to the assignables in the signature Σ are mobile.

Theorem 34.3 (Preservation for Typed Commands)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_\Sigma e : \tau$, *then* $\vdash_\Sigma e' : \tau$.
2. *If* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$, *with* $\vdash_\Sigma m \div \tau$ *and* $\mu : \Sigma$, *then* $\vdash_\Sigma m' \div \tau$ *and* $\mu' : \Sigma$.

Theorem 34.4 (Progress for Typed Commands)**.**

1. *If* $\vdash_\Sigma e : \tau$, *then either* e val_Σ, *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.
2. *If* $\vdash_\Sigma m \div \tau$ *and* $\mu : \Sigma$, *then either* $m \parallel \mu$ final_Σ *or* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ *for some* μ' *and* m'.

MA + Assignable References

A *reference* to an assignable a is a value, written &a, of *reference type* that determines the assignable a. A reference to an assignable provides the *capability* to get or set the contents of that assignable, even if the assignable itself is not in scope when it is used.

Two references can be compared for equality to test whether they govern the same underlying assignable. Two references that govern the same underlying assignable are *aliases*.

Reference types are compatible with both a scoped and a scope-free allocation of assignables. When assignables are scoped, the range of significance of a reference type is limited to the scope of the assignable to which it refers. Reference types are therefore immobile, so that they cannot be returned from the body of a declaration, nor stored in an assignable.

Supporting mutability requires that assignables be given a scope-free dynamics, so that their lifetime persists beyond the scope of their declaration. Consequently, all types are mobile, so that a value of any type may be stored in an assignable or returned from a command.

MA + Scoped Assignable References

Typ	τ	$::=$	$\mathtt{ref}(\tau)$	$\tau\,\mathtt{ref}$	assignable
Exp	e	$::=$	$\mathtt{ref}[a]$	$\&a$	reference
Cmd	m	$::=$	$\mathtt{getref}(e)$	$*\,e$	contents
			$\mathtt{setref}(e_1; e_2)$	$e_1 *= e_2$	update

The statics of reference types is defined by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{ref}[a] : \mathtt{ref}(\tau)} \tag{35.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{ref}(\tau)}{\Gamma \vdash_\Sigma \mathtt{getref}(e) \dot{\sim} \tau} \tag{35.1b}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{ref}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{setref}(e_1; e_2) \dot{\sim} \tau} \tag{35.1c}$$

MA + Scoped Assignable References

The dynamics of reference types defers to the corresponding operations on assignables, and does not alter the underlying dynamics of assignables:

$$\frac{}{\texttt{ref}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}} \tag{35.2a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{getref}(e) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{getref}(e') \parallel \mu} \tag{35.2b}$$

$$\frac{}{\texttt{getref}(\texttt{ref}[a]) \parallel \mu \underset{\Sigma,a\sim\tau}{\longmapsto} \texttt{get}[a] \parallel \mu} \tag{35.2c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\texttt{setref}(e_1; e_2) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{setref}(e_1'; e_2) \parallel \mu} \tag{35.2d}$$

$$\frac{}{\texttt{setref}(\texttt{ref}[a]; e) \parallel \mu \underset{\Sigma,a\sim\tau}{\longmapsto} \texttt{set}[a](e) \parallel \mu} \tag{35.2e}$$

MA + Scope-Free Assignable References

With immobile references it is impossible to build data structures containing references, or to return references from procedures.

To allow this, we must arrange that the lifetime of an assignable extend beyond its scope. In other words, we must give up stack allocation for heap allocation.

Assignables that persist beyond their scope of declaration are called *scope-free*, or just *free*, assignables. When all assignables are free, every type is mobile and so any value, including a reference, may be used in a data structure.

Supporting free assignables amounts to changing the dynamics so that allocation of assignables persists across transitions. We use transition judgments of the form

$$\nu\ \Sigma\ \{\, m \parallel \mu \,\} \longmapsto \nu\ \Sigma'\ \{\, m' \parallel \mu' \,\}.$$

MA + Scope-Free Assignable References

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\,\mathtt{ret}(e)\parallel\mu\,\}\ \mathsf{final}} \tag{35.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\,\mathtt{ret}(e)\parallel\mu\,\} \longmapsto \nu\ \Sigma\ \{\,\mathtt{ret}(e')\parallel\mu\,\}} \tag{35.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\,\mathtt{bnd}(e;x.m)\parallel\mu\,\} \longmapsto \nu\ \Sigma\ \{\,\mathtt{bnd}(e';x.m)\parallel\mu\,\}} \tag{35.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\,\mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e));x.m)\parallel\mu\,\} \longmapsto \nu\ \Sigma\ \{\,[e/x]m\parallel\mu\,\}} \tag{35.3d}$$

$$\frac{\nu\ \Sigma\ \{\,m_1\parallel\mu\,\} \longmapsto \nu\ \Sigma'\ \{\,m_1'\parallel\mu'\,\}}{\nu\ \Sigma\ \{\,\mathtt{bnd}(\mathtt{cmd}(m_1);x.m_2)\parallel\mu\,\} \longmapsto \nu\ \Sigma'\ \{\,\mathtt{bnd}(\mathtt{cmd}(m_1');x.m_2)\parallel\mu'\,\}} \tag{35.3e}$$

MA + Scope-Free Assignable References

$$\frac{}{\nu\ \Sigma, a \sim \tau\ \{\, \mathtt{get}[a] \parallel \mu \otimes a \hookrightarrow e \,\} \longmapsto \nu\ \Sigma, a \sim \tau\ \{\, \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{35.3f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{set}[a](e) \parallel \mu \,\} \longmapsto \nu\ \Sigma\ \{\, \mathtt{set}[a](e') \parallel \mu \,\}} \tag{35.3g}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a \sim \tau}}{\nu\ \Sigma, a \sim \tau\ \{\, \mathtt{set}[a](e) \parallel \mu \otimes a \hookrightarrow \_ \,\} \longmapsto \nu\ \Sigma, a \sim \tau\ \{\, \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{35.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{dcl}(e; a.m) \parallel \mu \,\} \longmapsto \nu\ \Sigma\ \{\, \mathtt{dcl}(e'; a.m) \parallel \mu \,\}} \tag{35.3i}$$

$$\frac{e\ \mathsf{val}_{\Sigma}}{\nu\ \Sigma\ \{\, \mathtt{dcl}(e; a.m) \parallel \mu \,\} \longmapsto \nu\ \Sigma, a \sim \tau\ \{\, m \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{35.3j}$$

MA + Scope-Free Assignable References

The language **RMA** extends **MA** with references to free assignables. Its dynamics is similar to that of references to scoped assignables given earlier.

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{getref}(e) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{getref}(e') \parallel \mu\,\}} \tag{35.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{getref}(\mathtt{ref}[a]) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{get}[a] \parallel \mu\,\}} \tag{35.4b}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\nu\,\Sigma\,\{\,\mathtt{setref}(e_1; e_2) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{setref}(e_1'; e_2) \parallel \mu\,\}} \tag{35.4c}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{setref}(\mathtt{ref}[a]; e_2) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{set}[a](e_2) \parallel \mu\,\}} \tag{35.4d}$$

MA + Scope-Free Assignable References

As an example of using **RMA**, consider the command $\texttt{newref}[\tau](e)$ defined by

$$\texttt{dcl}\, a := e \,\texttt{in}\, \texttt{ret}\,(\&a). \tag{35.5}$$

This command allocates a fresh assignable and returns a reference to it. Its static and dynamics are derived from the foregoing rules as follows:

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{newref}[\tau](e) \mathrel{\dot{\sim}} \texttt{ref}(\tau)} \tag{35.6}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\, \Sigma\, \{\, \texttt{newref}[\tau](e) \parallel \mu \,\} \longmapsto \nu\, \Sigma\, \{\, \texttt{newref}[\tau](e') \parallel \mu \,\}} \tag{35.7a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\, \Sigma\, \{\, \texttt{newref}[\tau](e) \parallel \mu \,\} \longmapsto \nu\, \Sigma, a \sim \tau\, \{\, \texttt{ret}(\texttt{ref}[a]) \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{35.7b}$$

Oftentimes, the command $\texttt{newref}[\tau](e)$ is taken as primitive, and the declaration command is omitted. In that case, all assignables are accessed by reference, and no direct access to assignables is provided.

MA + Scope-Free Assignable References

Although the proof of safety for references to scoped assignables presents few difficulties, the safety for free assignables is tricky. The main difficulty is to account for cyclic dependencies within data structures. The contents of one assignable may contain a reference to itself, or a reference to another assignable that contains a reference to it, and so forth. For example, consider the following procedure e of type $\mathtt{nat} \rightharpoonup \mathtt{nat\ cmd}$:

$$\mathtt{proc}\,(x : \mathtt{nat})\,\{\mathtt{if}\,(x)\,\mathtt{ret}\,(1)\,\mathtt{else}\,\{f \leftarrow @\,a\,;\, y \leftarrow f(x-1)\,;\, \mathtt{ret}\,(x * y)\}\}.$$

Let μ be a memory of the form $\mu' \otimes a \hookrightarrow e$ in which the contents of a contains, via the body of the procedure, a reference to a itself. Indeed, if the procedure e is called with a non-zero argument, it will "call itself" by indirect reference through a.

Cyclic dependencies complicate the definition of the judgment $\mu : \Sigma$. It is defined by the following rule:

$$\frac{\vdash_\Sigma m \div \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\,\Sigma\,\{m \parallel \mu\}\ \mathsf{ok}} \tag{35.8}$$

The first premise of the rule states that the command m is well-formed relative to Σ. The second premise states that the memory μ conforms to Σ, *relative to all of* Σ so that cyclic dependencies are permitted. The judgment $\vdash_{\Sigma'} \mu : \Sigma$ is defined as follows:

$$\frac{\forall a \sim \tau \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma} \tag{35.9}$$

MA + Scope-Free Assignable References

Theorem 35.1 (Preservation)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$, *then* $\vdash_\Sigma e' : \tau$.
2. *If* $\nu\ \Sigma\ \{ m \parallel \mu \}\ \mathsf{ok}$ *and* $\nu\ \Sigma\ \{ m \parallel \mu \} \longmapsto \nu\ \Sigma'\ \{ m' \parallel \mu' \}$, *then* $\nu\ \Sigma'\ \{ m' \parallel \mu' \}\ \mathsf{ok}$.

Theorem 35.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$, *then either* $e\ \mathsf{val}_\Sigma$ *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.
2. *If* $\nu\ \Sigma\ \{ m \parallel \mu \}\ \mathsf{ok}$ *then either* $\nu\ \Sigma\ \{ m \parallel \mu \}\ \mathsf{final}$ *or* $\nu\ \Sigma\ \{ m \parallel \mu \} \longmapsto \nu\ \Sigma'\ \{ m' \parallel \mu' \}$ *for some* Σ', μ', *and* m'.