Monotonic Typestates for the Object Calculus

Radha Jagadeesan[1], Alan Jeffrey[2], Corin Pitcher[1], and James Riely[1,⋆]

[1] School of Computing, DePaul University
[2] Bell Labs, Lucent Technologies

Abstract. Static analysis using typestates can ensure that dynamic object protocols are respected, for example that an object is accessed only after it is initialized. Typestate analyses often impose aliasing and linearity constraints, limiting their applicability to common practice. Monotone typestates avoid such constraints by limiting attention to properties that are stable under program dynamics, such as object initialization.

This paper presents a new approach to typing using monotone typestates. As in other recent work, our approach is expressive enough to address phased initialization protocols and the creation of cyclic data structures. These innovations allow for the elimination of `null` and the special status of constructors.

Our solution is more general than other published solutions, being the first that is able to statically validate the standard one-pass traversal algorithms for potentially cyclic graphs. These algorithms are both common and important. For example, one such algorithm is the basis for object deserialization.

1 Introduction

Traditional types capture the static signatures of functions and objects; for example, that field `f` references a file whereas field `n` references a node of a cyclic graph (when non-`null`). These types do not capture dynamically variant information, such that file `f` is open or closed or that `n` is initialized to be non-`null`. Traditional types provide an upper bound on capabilities available to an object; the type system always allows the calls `f.open()`, `f.close()` and `n.next()` regardless of the state of these fields. The more precise contract on the use of these capabilities is specified informally and tracked manually by programmers, creating many opportunities for error. Two distinct areas of research address this issue:

- Session types for communication centered programming (Takeuchi et al. 1994; Honda et al. 1998) specify the interaction between the sender and receiver on a channel as part of the type; the Singularity operating system (Hunt and Larus 2007) implements session types as communication abstractions in a general programming context. Static analysis is used to establish that no communication errors occur.
- Typestates for object protocols (Strom 1983; Strom and Yemini 1986) specify the structure of the interaction between the clients and the objects. For example, a typestate can distinguish between the open/closed state of a file or the uninitialized/initialized state of a data structure. Static analysis is used to establish that object protocols are respected.

⋆ Supported by NSF Career 0347542

Linearity and uniqueness ideas play a key technical role in both developments. In session types, linearity ensures that channels have unique senders and receivers at any instant, thus permitting the sender and receiver to agree on the current state of the session on the shared channel. The session type of a channel that does not satisfy these linearity conditions is forced to be a single-state entity, thus having *no* interesting state. In the object world, linearity is tantamount to unique references and allows one to sidestep the difficult analysis issues caused by aliasing, e.g., if there are multiple references to a socket object, how can two asynchronously executing clients agree on the typestate of the socket without explicit communication?

Such linearity and aliasing restrictions impede the widespread adoption of typestate ideas in general program development and motivate the consideration of monotone typestates. Monotone typestates (Fähndrich and Leino 2003) are stable under program dynamics, i.e., once a typestate of an object is established, no future interaction with the object (including imperative updates) invalidates this typestate assertion. Monotone typestates are amenable to relaxed aliasing constraints since various aliases to an object can only monotonically advance the typestate of the object (and other shared data structures). Furthermore, since the safe operation of a client on an object does not require awareness of the operations of other clients on shared objects, monotone typestates are also compatible with concurrency.

The main contribution of our paper is a formal development of a type system of monotone typestates for an imperative object calculus. We demonstrate that our type system is expressive enough to support phased initialization protocols that render specialized and ad hoc means such as constructors redundant. Thus, we facilitate incremental construction of objects (perhaps with cycles) even while ensuring that client access is restricted to properly initialized sections of the object reference graph. This is a feature that we share with the current state of the art (Qi and Myers 2009). *In addition*, our proposal is the first solution (to our knowledge) that is also able to statically validate the standard one-pass traversal algorithms for graphs (possibly with cycles).

Our contribution is particularly significant because the additional examples that our approach can handle are common and important. In Section 4 we show that we can accurately type the deserialization algorithm used in languages such as Java (Sun 2005). Serialization takes an object graph and turns it into stream of bytes that can be stored on disk or transmitted to another machine; deserialization reverses the process. Serialization is thus at the core of object permanence and distribution.

The type that we assign the deserialization algorithm establishes that the returned object graph is fully initialized, and therefore contains no `null` pointers. Such algorithms cannot be so typed using other published approaches.

Rest of this paper. In the remainder of this introduction, we briefly review Abadi and Cardelli's object calculus, which is the basis of our technical results, and present an overview of our approach. In Section 2, we describe the formalities of our variant of the object calculus and its type system. In Section 3 we present a series of basic examples which serve both to introduce the type system and to give some basic programming infrastructure upon which to build larger examples. At the end of Section 3 we include an encoding of workflows, and the discussion there delves a bit deeper into the workings of the type system. The heart of the paper is Section 4 which presents the deserialization

algorithm and its typing. An extended discussion of related and future work concludes the paper in Sections 5 and 6.

A review of the object calculus. We present our work using the (imperative) object calculus of Abadi and Cardelli (1996) because it is a linguistic "sweet spot" for formal discussions of graph algorithms: it has everything necessary to naturally describe such algorithms and nothing else.

The object calculus differs significantly from class-based languages: it has no classes, treats fields and methods uniformly, allows dynamic method update, and has explicit self parameters on methods but no primitive formal parameters for method arguments. We discuss these properties briefly as an introduction both to the basic ideas of the language and to the specifics of our variant.

The object calculus has two main syntactic categories: objects and terms. Objects are stored on a heap and manipulated via references. Object descriptions have the form $[\ell_1 = \varsigma(x_1)\, a_1, \ldots, \ell_n = \varsigma(x_n)\, a_n]$, representing an object with n members. Each member has a label ℓ and body $\varsigma(x)\, a$. In the body, x is the self parameter and a is a term. In class-based languages the self parameter is usually implicit, being represented as the reserved identifier `this` or `self`. The self parameter is indicated using the binder ς, in analogy with the formal parameter binder λ of the lambda calculus; as a result, the language is known as the sigma calculus. We discuss the encoding of formal parameters in Section 3.

As an example, suppose that heap reference p refers to the object $[\mathsf{left} = \varsigma(x)\, p, \mathsf{right} = \varsigma(x)\, p]$. This represents a minimal binary graph with a single node which is its own left and right child. Because the self parameter is not used in the bodies, we may think of left and right as fields. We elide the self binder on fields, preferring to write the object as $[\mathsf{left} = p, \mathsf{right} = p]$.

The basic operations of the language are member selection (read), member update (write) and sequencing (let). One writes $p.\mathsf{left}$ to access the value of p's left member and $p.\mathsf{left} \Leftarrow q$ to update it.

To observe the behavior of methods, let q reference the object $[\mathsf{val} = p, \mathsf{get} = \varsigma(x)\, x.\mathsf{val}]$. The selection $q.\mathsf{get}$ evaluates to p. The update $q.\mathsf{get} \Leftarrow (\varsigma(x)\, \mathsf{let}\, y = x.\mathsf{val};\, y.\mathsf{left})$ changes the get method of q so that it return $q.\mathsf{val}.\mathsf{left}$. Abadi and Cardelli show that method update allows for the natural encoding of method inheritance and overriding from class-based languages.

Our typing system assigns typenames to objects. The acceptable use of objects is managed by *constraints* on typenames, which one might also view as static *capabilities*. If $q : \psi$, then the constraint $\psi.\mathsf{val} : +\phi$ allows a program to select $q.\mathsf{val}$, with the result having typename ϕ. Similarly, the constraint $\psi.\mathsf{val} : -\phi$ allows a program to update $q.\mathsf{val}$ with any value of type ϕ. Typenames have exactly one object associated with them; therefore, the field is effectively immutable even with this capability. Mutability is allowed using existential quantifiers and type variables.

An overview of our approach. We develop formal foundations for a type system with monotone typestates. Our interfaces incorporate (monotone) typestates via pre and post conditions on access and updates to methods and fields. The monotonicity restriction means that once a field or method becomes available for use, it stays enabled for the rest

of the program's execution. A simple example is a logical variable (Lindstrom 1985) that moves from an uninitialized state (that does not have an enabled get method) to an initialized state (with a get method enabled) upon invocation of a set method.

It is noteworthy that we work in the context of a full imperative calculus in contrast to the constrained assignment in the treatment of Fähndrich and Leino (2003). Thus, in our setting, it is only the static reasoning via typestates that is constrained to be monotone.

In traditional object-oriented type systems, the type available to the client is a supertype of the actual object dynamic type. In our setting, this observation generalizes to also include the dynamically evolving typestate; i.e., any client of an object will always have a safe approximation to the actual typestate of the object which itself is a safe approximation of the currently available capabilities of the object. This invariant holds even with aliasing.

Our type system addresses the two challenges posed by Fähndrich and Xia (2007) and Fähndrich and Leino (2003): phased initialization and cyclic data structures.

With phased initialization we require flexibility in the construction and use of partially-initialized objects. To illustrate this requirement, consider the following program that constructs and initializes objects. The term fail is untypable (as defined at the beginning of Section 3); any attempt to access a field with value fail is a programming error.

(1) $\mathsf{let}\ x : \phi = [\mathsf{f} = \mathsf{fail}];$
(2) $\mathsf{let}\ y : \psi = [\mathsf{fst} = \mathsf{fail}, \mathsf{snd} = \mathsf{fail}];$
(3) $\mathsf{let}\ v : \tau = [\,];$
(4) $\mathsf{let}\ w : \rho = [\,];$
(5) $x.\mathsf{f} \Leftarrow y;$
(6) $x.\mathsf{f}.\mathsf{fst} \Leftarrow v;$
(7) $x.\mathsf{f}.\mathsf{snd} \Leftarrow w$

The assignment of the (at first uninitialized) pair y in $x.\mathsf{f} \Leftarrow y$ requires care to ensure that the type of $x.\mathsf{f}$ reflects the initialization status of y. In prior work, this has been dealt with by insisting that assignments use fully-initialized objects ("fully initialized" means at the top of a partial order on typestates). This prevents safety issues that are otherwise hard to avoid in the presence of aliasing, e.g., assigning a partially-initialized object to a field that already contains a fully-initialized object. However, it rules out the order of object initialization in the program above.

The type system in this paper accommodates a more general use of partially-initialized objects via forward references to type names that have not yet been generated. Such forward references to type names are used in types, e.g., in the above program, we assume the field update constraint $(\phi.\mathsf{f} :- \psi)$ at the creation of x. This constraint allows assignment to f of values of type ψ, a forward reference to the type name for y. Any run of the program will have at most one object with the type name ψ, and that is the only object that can be assigned to f. Thus, type names associated with objects are singleton types and the typing rule for "new" requires that a fresh name be used for the type of the freshly constructed object. With this property, assignments to the field can proceed (safely) without regard to the initialization status of the assigned object because multiple assignments always assign the same value. In particular, a field with a fully-initialized object cannot be overwritten with a partially-initialized object.

Moreover, the initialization status of object graphs is represented using additional constraints that accumulate monotonically. For example, after the assignment $x.\mathsf{f} \Leftarrow y$,

code is typechecked with the additional field access constraint $(\phi.\mathsf{f}:+\psi)$, meaning that a value of type ψ can be read from field f. Similarly, after the assignment $x.\mathsf{f}.\mathsf{fst} \Leftarrow v$, we add the constraint $(\psi.\mathsf{fst}:+\tau)$ (the existing constraints from the creation of y are $(\psi.\mathsf{fst}:-\tau, \psi.\mathsf{snd}:-\rho)$, which allow assignment to the fst and snd fields).

With regard to the second challenge of cyclic data structures, our type system permits the creation of cycles, e.g., pointing back to x from the second component of y:

(1) $\mathsf{let}\ x : \phi = [\mathsf{f} = \mathsf{fail}];$
(2) $\mathsf{let}\ y : \psi = [\mathsf{fst} = \mathsf{fail}, \mathsf{snd} = \mathsf{fail}];$
(3) $\mathsf{let}\ v : \tau = [\,];$
(4) $x.\mathsf{f} \Leftarrow y;$
(5) $x.\mathsf{f}.\mathsf{fst} \Leftarrow v;$
(6) $x.\mathsf{f}.\mathsf{snd} \Leftarrow x$

Here, we use the modified type constraints $(\psi.\mathsf{fst}:-\tau, \psi.\mathsf{snd}:-\phi)$ to type the creation of y, referring back to $x : \phi$ for the snd field. More generally, this analysis is as powerful as the recent clever methods that address initialization of strongly connected components (Qi and Myers 2009).

However, traversals of cyclic data structures, such as depth-first traversal, require further consideration. Static type systems can usually handle the straightforward inductive reasoning associated with tree traversal. These methods also work for dags *if* one permits recomputation on subgraphs, i.e., effectively unrolling the dag into a tree. Standard inductive reasoning fails if one attempts to statically validate the traversal of dags *without* recomputation or if one attempts to address the traversal of cyclic graphs because the children links at a node may point to nodes whose processing has begun but has not completed yet.

We show that our type system is expressive enough to statically validate the usual *single-pass* version of such a program without using any dynamic typestate tests.

A crucial role in this development is played by our method types whichlan have universal and existential quantification over typenames, as well as pre- and post-conditions. Significantly, pre-conditions are allowed to include existentially bound names.

The post-condition must be satisfied by the *callee*, who chooses the instantiation of the existential type variables, some of whom correspond to the internal implementation of the object.

The pre-condition must be satisfied by the *caller*, who chooses the instantiation of the universal type variables.

The inclusion of existentially bound variables in the pre-condition allows it to depend on internal object state. An external caller has little chance to establish such preconditions, but an internal caller may do better. In our static analysis of the graph traversal, we use such types for " internal implementation methods" that are only invoked as helper methods from other internal methods of the object. These internal methods do have knowledge of the object state. This form of quantification is key to establishing the invariants required for graph traversal.

2 An imperative object calculus

We investigate typestates using a variant of Abadi and Cardelli's (1996) imperative ς-calculus. In this presentation we elide cloning, although we expect it could be handled with little additional complexity.

We presuppose disjoint sets for *field names* (ranged over by f), *method names* (m), *object names* (p, q), *object variables* (x, y, z), *type names* (ϕ, ψ), *type variables* (α, β, γ) and *type recursion variables* (θ). Object names are the values of the language. *Member names*, *object identifiers* and *type identifiers* are defined as follows.

$$\ell ::= f \mid m \qquad \text{(Member names)}$$
$$v, w ::= p \mid x \qquad \text{(Object identifiers)}$$
$$\rho, \tau ::= \phi \mid \alpha \qquad \text{(Type identifiers)}$$

For all syntax categories, we define the functions *fn*, *fv*, and *fid* to return the sets of free object and type names, variables, and identifiers, respectively (these do not include member names, which have no binders). We identify syntax up to renaming of bound identifiers. For any syntax category ζ, we write $\zeta\{\vec{v}/\vec{x}\}$ for the capture avoiding substitution of $\vec{v}$ for $\vec{x}$ in ζ, and similarly for $\zeta\{\vec{\tau}/\vec{\alpha}\}$ and $\zeta\{\vec{A}/\vec{\theta}\}$. The syntax of objects and terms are as follows.

$$o, n ::= [\ell_1 = \varsigma(x_1)\, a_1, \ldots, \ell_n = \varsigma(x_n)\, a_n] \qquad \text{(Objects)}$$
$$a, b ::= o \mid v \mid v.\ell \mid v.\ell \Leftarrow \varsigma(x)\, a \mid \mathsf{let}\, x = a; b \qquad \text{(Terms)}$$

An *abstraction* "$\varsigma(x)\,a$" includes the *self binder* x, with scope a. We elide the binder in a *field value* of the form "$\varsigma(x)\,v$" where $x \neq v$; thus an object with one field may be written $[f = v]$ as shorthand for $[f = \varsigma(x)\, v]$.

An *object* is a record of labeled abstractions (succinctly $[\vec{\ell} = \varsigma(\vec{x})\,\vec{a}]$). If ℓ is a field name, then we require that the corresponding abstraction be a field value. In object $[\vec{\ell} = \varsigma(\vec{x})\,\vec{a}]$, we require that $\vec{\ell}$ be distinct and identify objects up to reordering of declarations. We define the *domain* of an object as $dom([\vec{\ell} = \varsigma(\vec{x})\,\vec{a}]) \triangleq \{\vec{\ell}\}$.

Terms include object declarations, identifiers, selection and update. We require that fields be updated only with field values. The let construct binds x with scope b; we use standard syntax sugar for sequencing, for example writing "$a.f \Leftarrow v$" for "$\mathsf{let}\, x = a; x.f \Leftarrow v$" where $x \neq v$.

A *store* (S) is a sequence of name/object pairs ($S ::= p_1 = o_1, \ldots, p_n = o_n$). Similar to objects, we require that the names $\vec{p}$ be distinct and identify stores up to reordering. Evaluation relates *configurations*, which consist of a store and a term ($S \mathbin{/\!/} a \to S' \mathbin{/\!/} a'$).

Evaluation (where $S = (S', p = [\vec{\ell} = \varsigma(\vec{x})\,\vec{a}, \ell = \varsigma(x)\, a]))$

$S \mathbin{/\!/} o \to S, q = o \mathbin{/\!/} q$	if $q \notin dom(S)$
$S \mathbin{/\!/} p.\ell \to S \mathbin{/\!/} a\{^p/_x\}$	
$S \mathbin{/\!/} p.\ell \Leftarrow \varsigma(y)\, b \to (S', p = [\vec{\ell} = \varsigma(\vec{x})\,\vec{a}, \ell = \varsigma(y)\, b]) \mathbin{/\!/} p$	
$S \mathbin{/\!/} \mathsf{let}\, x = p; b \to S \mathbin{/\!/} b\{^p/_x\}$	
$S \mathbin{/\!/} \mathsf{let}\, x = a; b \to S' \mathbin{/\!/} \mathsf{let}\, x = a'; b$	if $S \mathbin{/\!/} a \to S' \mathbin{/\!/} a'$

In contrast with Abadi and Cardelli, our semantics uses substitutions rather than stacks, let expressions rather than evaluation contexts, and object stores rather than closure stores.

Typing

Syntax

$$A,B ::= \exists(\tau_1,\ldots,\tau_k)\{P_1,\ldots,P_m\}\tau[Q_1,\ldots,Q_n] \mid \theta \mid \boldsymbol{\mu}(\theta)A \qquad \textit{(Term types)}$$

$$F,G ::= \forall(\alpha_1,\ldots,\alpha_n)A \qquad \textit{(Method types)}$$

$$P,Q,R ::= \mathbf{0} \mid \rho{\equiv}\tau \mid *{:}A \mid \tau.\ell{:}\chi F \qquad \textit{(Constraints)}$$

$$\chi ::= - \mid + \mid \times \qquad \textit{(Annotations)}$$

$$E ::= \cdot \mid E,\tau \mid E,v{:}\rho \mid E,P \qquad \textit{(Environments)}$$

$$(\{\tau,v\} \cap dom(E) = \emptyset,\ \{\rho\} \cup fn(P) \subseteq dom(E))$$

$$\mathsf{Top} \triangleq \exists(\alpha)\{\mathbf{0}\}\alpha \qquad \textit{(Top type)}$$

Typing $(E \vdash a : A)$

(T-OBJ)
$$\frac{\forall((\phi.\ell{:}-F) \in \vec{P})\ \ \forall((\phi.\ell{:}\chi G) \in \vec{P})\ \ E \vdash F \leqslant G \quad \forall((\phi.\ell{:}+\forall(\vec{\alpha})A) \in \vec{P})\ \ \exists((\ell = \varsigma(x)a) \in o)\ \ E,\phi,\vec{\alpha},x{:}\phi,\vec{P} \vdash a : A}{E \vdash o : \exists(\phi)\phi[\vec{P}]} \quad \begin{array}{l} dom(o) = \{\vec{\ell}\} \\ \vec{P} = (\phi.\vec{\ell}{:}\vec{\chi}\vec{F}) \end{array}$$

(T-ID)
$$\frac{(v{:}\tau) \in E}{E \vdash v : \tau}$$

(T-SEL)
$$\frac{E \vdash v : \tau[\tau.\ell{:}+A]}{E \vdash (v.\ell) : A}$$

(T-UPD)
$$\frac{E \vdash v : \tau[\tau.\ell{:}-A] \qquad E,x{:}\tau \vdash a : A}{E \vdash (v.\ell \Leftarrow \varsigma(x)a) : \tau[\tau.\ell{:}+\mathsf{Top}]}$$

(T-LET)
$$\frac{E,\vec{\tau} \vdash a : \exists(\vec{\rho})\{\vec{P}\}\rho[\vec{Q}] \qquad E,\vec{\rho},x{:}\rho \vdash b : \exists(\vec{\tau})\{\vec{P},\vec{Q}\}\tau[\vec{R}]}{E \vdash (\mathsf{let}\ x = a; b) : \exists(\vec{\tau},\vec{\rho})\{\vec{P}\}\tau[\vec{Q},\vec{R}]}$$

Subtyping $(E \vdash A \leqslant B \text{ and } E \vdash F \leqslant G)$

(U-TRM)
$$\frac{\forall(P \in \vec{P})\ \ E,\vec{\phi},\vec{\psi},\vec{\alpha},\vec{R}\{\vec{\tau}/\vec{\beta}\} \vdash P \qquad \forall(S \in \vec{S})\ \ E,\vec{\phi},\vec{\psi},\vec{\alpha},\vec{Q},\vec{R}\{\vec{\tau}/\vec{\beta}\} \vdash S\{\vec{\tau}/\vec{\beta}\}}{E \vdash (\exists(\vec{\phi},\vec{\psi},\vec{\alpha})\{\vec{P}\}\rho[\vec{Q}]) \leqslant (\exists(\vec{\phi},\vec{\beta})\{\vec{R}\}\tau[\vec{S}])} \quad \rho = \tau\{\vec{\tau}/\vec{\beta}\}$$

(U-MTH)
$$\frac{E,\vec{\beta} \vdash (A\{\vec{\tau}/\vec{\alpha}\}) \leqslant B}{E \vdash (\forall(\vec{\alpha})A) \leqslant (\forall(\vec{\beta})B)}$$

(U-UNROLL-LEFT)
$$\frac{E \vdash A\{\boldsymbol{\mu}(\theta)A/\theta\} \leqslant B}{E \vdash \boldsymbol{\mu}(\theta)A \leqslant B}$$

(U-UNROLL-RIGHT)
$$\frac{E \vdash A \leqslant B\{\boldsymbol{\mu}(\theta)B/\theta\}}{E \vdash A \leqslant \boldsymbol{\mu}(\theta)B}$$

(U-TOP)
$$\frac{}{E \vdash A \leqslant \mathsf{Top}}$$

Derivation $(E \vdash P)$

(D-ID)
$$\frac{P \in E}{E \vdash P}$$

(D-EQ)
$$\frac{}{E \vdash \tau{\equiv}\tau}$$

(D-NEQ)
$$\frac{E \vdash \phi{\equiv}\psi}{E \vdash \mathbf{0}}\ \phi \neq \psi$$

(D-INH)
$$\frac{E \vdash v : A}{E \vdash *{:}A}$$

(D-RD-FRD)
$$\frac{E \vdash \tau.\ell{:}\times G \qquad E \vdash \tau.\ell{:}+F}{E \vdash \tau.\ell{:}+G}$$

Subsumption rules $(\text{where } \{\vec{R}\}(\exists(\vec{\tau})\{\vec{P}\}\tau[\vec{Q}]) \triangleq (\exists(\vec{\tau})\{\vec{R},\vec{P}\}\tau[\vec{Q}]) \text{ when } \vec{\tau} \text{ disjoint } \mathit{fid}(\vec{R}))$

(T-SUB-PRE)
$$\frac{E,\vec{P} \vdash a : A}{E \vdash a : \{\vec{P}\}A}$$

(T-SUB-TRM)
$$\frac{E \vdash a : A \qquad E \vdash A \leqslant B}{E \vdash a : B}$$

(D-SUB-INH)
$$\frac{E \vdash *{:}A \qquad E \vdash A \leqslant B}{E \vdash *{:}B}$$

(D-SUB-WR)
$$\frac{E \vdash \tau.\ell{:}-F \qquad E \vdash G \leqslant F}{E \vdash \tau.\ell{:}-G}$$

(D-SUB-RD)
$$\frac{E \vdash \tau.\ell{:}+F \qquad E \vdash F \leqslant G}{E \vdash \tau.\ell{:}+G}$$

(D-SUB-FRD)
$$\frac{E \vdash \tau.\ell{:}\times F \qquad E \vdash F \leqslant G}{E \vdash \tau.\ell{:}\times G}$$

Structural rules $(\text{where } \mathscr{J} ::= a : A \mid A \leqslant B \mid F \leqslant G \mid P)$

(S-FALSE)
$$\frac{E \vdash \mathbf{0}}{E \vdash \mathscr{J}}$$

(S-EQ)
$$\frac{E\{^{\tau}/_{\alpha}\} \vdash \mathscr{J}\{^{\tau}/_{\alpha}\}}{\alpha,E,\alpha{\equiv}\tau \vdash \mathscr{J}}$$

(S-CUT)
$$\frac{E \vdash *{:}\tau[\tau.f{:}+A] \qquad E,*{:}A \vdash \mathscr{J}}{E \vdash \mathscr{J}}$$

Types. The typing system is presented on the following page, although we expect that at first reading only the first four lines, which present the syntax of types, are essential.

As sketched at the end of our introduction to the object calculus, our system assigns typenames to identifiers. The capabilities available on a given identifier are controlled by *constraints* placed on the corresponding typename.

In addition to the type of the result, term types (A, B) include pre- and post-conditions with existential quantifiers. The term type "$\exists(\vec{\tau})\{\vec{P}\}\tau[\vec{Q}]$" may be assigned to a term that produces a value of type τ satisfying postconditions $\vec{Q}$, for some choice of type identifiers $\vec{\tau}$ which satisfy the preconditions $\vec{P}$. ($\vec{\tau}$ are bound with scope $\vec{P}$, τ and $\vec{Q}$.) We allow existential quantification over both variables and names: quantification over variables is traditional; quantification over names corresponds to the ∇ binder of Miller and Tiu (2005). Quantification over names allows us to type a standard prelude for booleans in Section 3. Thus $\exists(\phi)\exists(\psi)\phi{\equiv}\psi$ is always false, whereas $\exists(\phi)\exists(\alpha)\phi{\equiv}\alpha$ is always true.

Top is the top type for terms; we discuss this after presenting subtyping below. Recursive term types $\boldsymbol{\mu}(\theta)A$ are handled by unrolling. We require that term types be closed with respect to type recursion variables.

Method types (F, G) also allow universal quantification. Type "$\forall(\vec{\alpha})A$" may be assigned to a method that behaves as term A for any $\vec{\alpha}$. ($\vec{\alpha}$ are bound with scope A.)

Types are identified up to renaming of bound identifiers; types and environments are identified up to reordering. We elide empty quantifiers and pre/post-conditions.

Pre- and post-conditions are sets of *constraints* (P, Q).

The *unsatisfiable constraint* "**0**" can be encoded true$\equiv$false in the presence of the standard prelude defined in Section 3.

An *equality constraint* "$\rho{\equiv}\tau$" may be used to derive contradictions using the derivation rule D-NEQ along with the structural rule S-FALSE.

An *inhabitedness constraint* "$*{:}A$" is useful in inductive proofs, such as the graph examples presented in Section 4; the structural rule S-CUT expresses propagation of inhabitedness through fields.

A *member constraint* "$\tau.\ell{:}\chi F$" states that objects associated with type identifier τ have a member ℓ of type F with *annotation* χ. The contravariant *write* annotation ($-$) and covariant *read* annotation ($+$) are familiar from (Abadi and Cardelli 1996). The covariant *future read* ($\times$) annotation is new. D-FRD-RD states that if a member has a future read constraint at type G and it is readable at any type, then it is readable at G. Future reads are useful in conjunction with update; T-UPD states that once a term has been written, it will always receive a read capability at some type.

Typing. The judgment $E \vdash a : A$ indicates that a has term type A under environment E. This makes use of auxiliary judgments for subtyping ($E \vdash A \leqslant B$ and $E \vdash F \leqslant G$) and entailment of constraints ($E \vdash P$). In our presentation, we separate the inductive rules from subsumption and structural rules.

The rule T-OBJ for object creation requires that the new object o have a fresh typename ϕ and that any constraints $\vec{P}$ placed on this typename must be consistent with each other and with the definition of the members of o. The freshness of ϕ follows from the meaning of the ∇ binder (Miller and Tiu 2005) as discussed above.

The rules for identifiers and selection are standard. As usual, T-SEL requires a read annotation on $\tau.\ell$, whereas T-UPD places the symmetric requirement for update. In addition, T-UPD adds a new constraint to the existing type τ; after the update, $\tau.\ell$ becomes readable at type Top. Although Top is a useless type by itself, one may use D-RD-FRD, in conjunction with a future read annotation on $\tau.\ell$, to convert this useless read capability into one which is useful.

This formulation of T-UPD allows covariant reads and contravariant writes. If the conclusion of T-UPD were to allow the member to be read at type A, then at least one of these must become invariant in order to preserve soundness.

T-LET supports "C style forwarding" of types by allowing let to be recursive on existential types. In particular, note that in "let $x = a$; b", the type of a may include type variables that are existentially bound in the type of b. The forwarding ability of let can be used to assign

$$\mathsf{let}\, x = [\mathsf{f} = \mathsf{fail}];\ \mathsf{let}\, x' = [\mathsf{f} = x];\ x.\mathsf{f} \Leftarrow x';\ [\mathsf{m} = x,\ \mathsf{n} = x']$$

the type

$$\exists(\phi, \psi, \psi')\, \phi\, [\phi.\mathsf{m} :+ \psi,\ \phi.\mathsf{n} :+ \psi',\ \psi.\mathsf{f} :+ \psi',\ \psi'.\mathsf{f} :+ \psi].$$

The resulting structure is circular, and this circularity is manifest in its type.

The subtyping rules allow one to instantiate quantifiers in the standard way (by choosing $\vec{\tau}$). In addition, they formalize the idea that subtypes demand less (so fewer preconditions) and give more (so more postconditions). Thus we have $\vdash \tau \leqslant \{P\}\tau$ and $\vdash \tau[Q] \leqslant \tau$. Inferences in the other direction can be achieved in the presence of a sufficiently powerful environment. Thus we have $\vec{P} \vdash \{\vec{P}\}\tau \leqslant \tau$ and $\vec{Q} \vdash \tau \leqslant \tau[\vec{Q}]$.

Note that $\exists(\alpha, \phi, \psi)\{\phi \equiv \psi\}\alpha$ is not a top type since the rules for subtyping allow for the removal of existentially bound names in subtypes rather than supertypes. In the presence of the standard prelude described in Section 3, we can define the top type as $\exists(\alpha)\{\mathsf{true} \equiv \mathsf{false}\}\alpha$.

The rules for derivation are straightforward. D-INH allows one to conclude that a type is inhabited if there is a witness. D-RD-FRD converts future reads into reads, as already discussed. The subsumption rules are straightforward as well. Note that future read capabilities are covariant.

D-NEQ, combined with S-FALSE corresponds to the (intuitionist) reading of $\phi \neq \psi$ as the deduction of falsity from $\phi \equiv \psi$. The reserved name fail has no typing rules, and thus can only be typed by contradiction using this second rule. As we will see in the examples, this rule will be invoked when attempting to equate distinct singleton reference types, such as those corresponding to distinct boolean values. The structural rule S-EQ expresses that equality is substitutive. S-CUT expresses propagation of inhabitedness through fields: if τ is inhabited and it permits read-access to a field, then since fields are only written with values, the type of the field is inhabited. S-CUT is required to typecheck the cyclic graph example presented in Section 4.

Properties of typing. Whereas terms may generally have undischarged preconditions, we require that top level term, which we call *programs*, do not.

Definition (Program). A term a is a (well typed) *program* if $\cdot \vdash a : \exists(\alpha)\,\alpha$.

Following Wright and Felleisen (1994), our main result is *weak safety* which states that well typed programs cannot get *stuck*; that is, a well typed program will either diverge or yield a value.

Theorem (Weak Safety). Let a be a well typed program and suppose $\cdot /\!\!/ a \to^* S /\!\!/ b$. Then either $b = p$ (for some $p \in S$) or $S /\!\!/ b \to S' /\!\!/ b'$ (for some S' and b').

The theorem fails for general terms which may have undischarged preconditions. The proof of weak safety requires auxiliary judgments for well-formed configurations, stores and environments, which we elide.

A proof sketch can be found in Appendix A. We discuss decidability issues in Section 6.

3 Examples

Untouchable objects. We say that a term is *untypable* if can only be typed using S-FALSE. Note that S-FALSE cannot be applied to top level terms that satisfy the conditions for weak safety: empty environment and no preconditions.

The examples use the distinguished object name fail to represent an untouchable object, such as the initial state of $x.\mathsf{f}$ in

$$\mathsf{let}\, x = [\mathsf{f} = \mathsf{fail}]; x.\mathsf{f} \Leftarrow v; x.\mathsf{f}.$$

Formally, fail is not allowed to appear in the domain of the top level environment E, so is untypable, and hence by type safety untouchable.

Because we follow Abadi and Cardelli's convention that a method must be present in order to overridden, we use fail to initialize untouchable fields, rather than simply eliding the fields.

Self types. Self types and type bounds on type variables, in the style of Abadi and Cardelli's (1996) imperative ς-calculus with Self types, are not explicit in our presentation, but can be encoded as follows

$$\varsigma(\alpha)\,[\vec{P}] \triangleq \exists(\alpha)\,\alpha\,[\vec{P}]$$
$$\tau{\leq}\varsigma(\alpha)\,[\vec{P}] \triangleq \vec{P}\{^{\tau}/_{\alpha}\}$$

Here $\varsigma(\alpha)\,[\vec{P}]$ is a term type and $\tau{\leq}\varsigma(\alpha)\,[\vec{P}]$ is a constraint. The notational similarity between $\leq$ and $\leqslant$ is intentional. The constraint judgment $E \vdash \tau{\leq}\varsigma(\alpha)\,[\vec{P}]$ holds if and only if the subtyping judgment $E \vdash \tau \leqslant \varsigma(\alpha)\,[\vec{P}]$ holds.

To contain the scope of existential binders, we sometimes extend this convention. For positively occuring constraints, such as those in top-level postconditions, we allow additional existential bounds with the interpretation that the binder is to be pulled to the closest available scope. Thus, if $T \triangleq \exists(\vec{\rho})\varsigma(\alpha)\,[\vec{P}]$, we may write "$\beta.m\colon{+}\gamma[\tau{\leq}T]$" as shorthand for "$\beta.m\colon{+}\exists(\vec{\rho})\,\gamma[\vec{P}\{^{\tau}/_{\alpha}\}]$".

Functions. Following Abadi and Cardelli, we define derived forms for function abstraction and application

$$A \to B \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{arg}:-A,\ \alpha.\mathsf{val}:+\{\alpha.\mathsf{arg}:+\mathsf{Top}\}B]$$
$$\lambda(x)\,b \triangleq [\mathsf{arg}=\mathsf{fail},\ \mathsf{val}=\varsigma(y)\,(\mathsf{let}\ x=y.\mathsf{arg};\ b)]$$
$$v(w) \triangleq (v.\mathsf{arg} \Leftarrow w;\ v.\mathsf{val})$$

with the derived typing and subtyping rules

$$\frac{E, x{:}A \vdash b : B}{E \vdash \lambda(x)\,b : A \to B} \qquad \frac{E \vdash v : A \to B \quad E \vdash w : A}{E \vdash v(w) : B} \qquad \frac{E \vdash A' \leqslant A \quad E \vdash B \leqslant B'}{E \vdash (A \to B) \leqslant (A' \to B')}$$

where the auxiliary judgement $E, x{:}A \vdash b : B$ is defined as

$$\frac{E, \vec{\tau}, v{:}\tau, \vec{P}, \vec{Q} \vdash b : C \quad E \vdash \exists(\vec{\tau})\,\{\vec{P}\}C \leqslant B}{E, v{:}(\exists(\vec{\tau})\,\{\vec{P}\}\tau\,[\vec{Q}]\,) \vdash b : B}$$

Our treatment of abstraction, is more precise than that of Abadi and Cardelli, who set "arg = diverge" initially, allowing that if val is called before arg is set then the term will diverge. Note that fail is untypable, so our type system ensures that arg must be set before val is called.

Since fail is untypable, the initial assignment to arg must be ignored by the type rules. Indeed this is the case, since T-OBJ places constraints only on readable fields and arg is not initially readable. The read of arg in val is allowed only because of the precondition $\{\alpha.\mathsf{arg}:+\mathsf{Top}\}$; initially, val will not typecheck without this precondition.

The type given above for abstractions is the *external* type, required for application. To type abstractions themselves requires the *internal* type

$$\varsigma(\alpha)\,[\alpha.\mathsf{arg}:\times A,\ \alpha.\mathsf{val}:+\{\alpha.\mathsf{arg}:+\mathsf{Top}\}B]\,.$$

After an update to arg, the external type of the function is

$$\varsigma(\alpha)\,[\alpha.\mathsf{arg}:-A,\ \alpha.\mathsf{arg}:+\mathsf{Top},\ \alpha.\mathsf{val}:+\{\alpha.\mathsf{arg}:+\mathsf{Top}\}B]$$

which is a subtype of

$$\varsigma(\alpha)\,[\alpha.\mathsf{arg}:-A,\ \alpha.\mathsf{arg}:+\mathsf{Top},\ \alpha.\mathsf{val}:+B]$$

at which point val may be called, with the expectation that it will return a result of type B. The internal type of the function is then

$$\varsigma(\alpha)\,[\alpha.\mathsf{arg}:+\mathsf{Top},\ \alpha.\mathsf{arg}:\times A,\ \alpha.\mathsf{val}:+B]$$

which is a subtype of

$$\varsigma(\alpha)\,[\alpha.\mathsf{arg}:+A,\ \alpha.\mathsf{val}:+B]$$

thus allowing val to be read in the function body without imposing any precondition on the surrounding context.

By convention, preconditions bind as weakly as possible and postconditions bind as strongly as possible. Thus, "$\{\vec{P}\}A \to B\,[\vec{Q}]$" should be read "$\{\vec{P}\}(A \to (B\,[\vec{Q}]\,))$".

Immutable pairs. Pairs can be encoded as one would expect.

$$(v,w) \triangleq [\mathsf{fst}=v, \mathsf{snd}=w]$$
$$\lambda(x,y)\,a \triangleq \lambda(z)\,\mathsf{let}\ x=z.\mathsf{fst};\ \mathsf{let}\ y=z.\mathsf{snd};\ a \quad (\text{where } z \notin \mathit{fid}(a))$$

Consider the first term below.

$$\mathsf{let}\ x=[\mathsf{f}=\mathsf{fail}];\ \mathsf{let}\ y=[\mathsf{fst}=x,\ \mathsf{snd}=x];\ (y.\mathsf{fst}).\mathsf{f} \Leftarrow \mathsf{r};\quad \ldots \quad (y.\mathsf{fst}).\mathsf{f}$$
$$\mathsf{let}\ x=[\mathsf{f}=\mathsf{fail}];\ \mathsf{let}\ y=[\mathsf{fst}=x,\ \mathsf{snd}=x];\ (y.\mathsf{fst}).\mathsf{f} \Leftarrow \mathsf{r};\ y.\mathsf{fst} \Leftarrow [\mathsf{f}=\mathsf{fail}];\ (y.\mathsf{fst}).\mathsf{f}$$

While the term appears to safe, this is only the case if $y.\mathsf{fst}$ is immutable. As the second term demonstrates, an update to $y.\mathsf{fst}$ causes a runtime failure to occur. One would like to type the pair in such a way that first term is typable but the second is not. We introduce the constraint $\rho.\ell:\boxplus F$ for *immutable* fields, similar to Java's "final" annotation, defined as

$$\rho.\ell:\boxplus\forall(\vec{\tau})\exists(\vec{\beta})\{\vec{P}\}\tau[\vec{Q}] \triangleq \rho.\ell:+\forall(\vec{\tau})\exists(\vec{\beta})\{\vec{P}\}\tau[\vec{Q},\rho.\ell:+\tau].$$

The annotation simply adds a postcondition: after a value of type τ is read, only values of type τ may be read. Using the immutable pair type

$$A \times B \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{fst}:\boxplus A,\ \alpha.\mathsf{snd}:\boxplus B]$$

the first program above is typable, whereas the second is not: The immutability annotation forces all of the values assigned to $y.\mathsf{fst}$ to have the same typename, say ϕ. Since $y.\mathsf{fst}$ is read in the third line, ϕ must have a bound that includes a type for f which must be compatible with r. The object on the fourth line can not be assigned the same typename because it does not satisfy the bound.

Booleans. For any term a, the following *standard prelude* generates the boolean values, binding them to the reserved variables true and false.

$$\mathsf{let\ true}=[\mathsf{case}=\lambda(x)x.\mathsf{istrue},\ \mathsf{assert}=\varsigma(x)x,\ \mathsf{nassert}=\mathsf{fail}];$$
$$\mathsf{let\ false}=[\mathsf{case}=\lambda(x)x.\mathsf{isfalse},\ \mathsf{assert}=\mathsf{fail},\ \mathsf{nassert}=\varsigma(x)x];\ a$$

true.case calls back on its argument's istrue method, whereas false.case calls back on isfalse. The methods false.assert and true.nassert will fail, but it follows from Weak Safety that well-typed assertions cannot fail. The conditional is encoded as follows.

$$\begin{array}{l}\mathsf{if}\ w\ \mathsf{then}\ a\ \mathsf{else}\ b \triangleq \mathsf{let}\ v=[\mathsf{result}=\mathsf{fail},\ \ \mathsf{done}=\mathsf{false},\\ \qquad\qquad\qquad\qquad \mathsf{istrue}=\varsigma(x)\,\mathsf{let}\ y=a;\ x.\mathsf{result}\Leftarrow y;\ x.\mathsf{done}\Leftarrow\mathsf{true};\ x,\\ \qquad\qquad\qquad\qquad \mathsf{isfalse}=\varsigma(x)\,\mathsf{let}\ y=b;\ x.\mathsf{result}\Leftarrow y;\ x.\mathsf{done}\Leftarrow\mathsf{true};\ x];\\ \qquad\qquad\qquad w.\mathsf{case}(v);\ v.\mathsf{done}.\mathsf{assert};\ v.\mathsf{result}\end{array}$$

It is worth noting that this is an example of the standard implementation of the *visitor pattern* in imperative languages (Gamma et al. 1994). In order to provide memory safety without dynamic checks, one must usually abandon this idiom in favor of the functional style used in ML case expressions. We will show later how this idiom can

be typechecked using typestates. Note that done is as a "compile time" field, with no dynamic significance; its entire purpose is to communicate the fact that the visitor has been visited. The assertion $v.\mathsf{done}.\mathsf{assert}$ is guaranteed to succeed in well typed code.

The type rule T-OBJ generates new typenames for true and false, which by convention we will assume to be True and False. We define the *standard environment* as (we define *True* and *False* below)

$$\mathsf{True}, \mathsf{False}, \mathsf{true}:\mathsf{True}, \mathsf{false}:\mathsf{False}, \mathsf{True}\leq \mathit{True}, \mathsf{False}\leq \mathit{False}.$$

Typing the standard prelude as a *program* (not just a term), requires existential quantification over these typenames. *True* is defined as follows (*False* exchanges the types for assert and nassert; *FalseV* replaces istrue with isfalse)

$$\begin{array}{l}\mathit{True} \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{case}:+\forall(\beta)\{\beta\leq \mathit{TrueV}\}\beta\to\beta\,[\beta.\mathsf{done}:+\mathsf{True}]\,,\\ \qquad\qquad \alpha.\mathsf{assert}:+\mathsf{True},\\ \qquad\qquad \alpha.\mathsf{nassert}:+\forall(\beta)\{\mathsf{True}\equiv\mathsf{False}\}\beta]\\ \mathit{TrueV} \triangleq \varsigma(\beta)\,[\beta.\mathsf{istrue}:+\beta\,[\beta.\mathsf{done}:+\mathsf{True}]]\end{array}$$

In order to allow nassert to be readable in the supertype *Bool*, we must give a type to the method, but the body of nassert is fail, which is untypable. We can not use the same trick that we did in typing function application, since nassert is always readable and never updated. Instead, the precondition True≡False indicates that, although present, the field may never be accessed. The precondition, combined with T-PRE and STRUCT-NEQ, allows the body of nassert to type trivially. *True* and *False* have the common supertype *Bool*.

$$\begin{array}{l}\mathit{Bool} \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{case}:+\forall(\beta)\{\beta\leq \mathit{BoolV}(\alpha)\}\beta\to\beta\,[\beta.\mathsf{done}:+\mathsf{True}]\,,\\ \qquad\qquad \alpha.\mathsf{assert}:+\{\alpha\equiv\mathsf{True}\}\alpha,\\ \qquad\qquad \alpha.\mathsf{nassert}:+\{\alpha\equiv\mathsf{False}\}\alpha]\\ \mathit{BoolV}(\alpha) \triangleq \varsigma(\beta)\,[\beta.\mathsf{istrue}:+\{\alpha\equiv\mathsf{True}\}\beta\,[\beta.\mathsf{done}:+\mathsf{True}]\,,\\ \qquad\qquad\qquad \beta.\mathsf{isfalse}:+\{\alpha\equiv\mathsf{False}\}\beta\,[\beta.\mathsf{done}:+\mathsf{True}]]\end{array}$$

The fact that $\mathit{BoolV}(\mathsf{True}) \leqslant \mathit{TrueV}$ is used in proving $\mathsf{True} \leqslant \mathit{Bool}$. The precondition of nassert in *Bool* must be weaker than that of *True* since the nassert may be called on *False*. Fortunately, the precondition $\alpha\equiv$False is is identical to True≡False when typing inhabitants of True. The conditional has the following derived type rule.

$$\frac{E\vdash w:\tau \quad E\vdash \tau\leq \mathit{Bool} \quad E,\tau\equiv\mathsf{True}\vdash a:A \quad E,\tau\equiv\mathsf{False}\vdash b:A}{E\vdash \mathsf{if}\ w\ \mathsf{then}\ a\ \mathsf{else}\ b:A}$$

Deriving this typing rule is non-trivial, as the field access $v.\mathsf{result}$ must be shown to be safe, even though $v.\mathsf{result}$ is initialized to fail. The safety depends on using the field done to track whether result has been set, and moreover requires that once done is set true, it is never set false again, thus allowing result to be accessed. This is an example of a monotone boolean with a constraint, which we now discuss.

Booleans with constraints. For any constraints $\vec{P}$, define the object type "$\mathsf{True}{\Rightarrow}\vec{P}$" as follows.

$$\begin{array}{rl}\mathsf{True}{\Rightarrow}\vec{P} \triangleq & \varsigma(\alpha)\,[\alpha.\mathsf{case}:{+}\forall(\beta)\{\beta{\leq}\mathit{BoolV}(\alpha)\}\beta\to\beta\,[\beta.\mathsf{done}:{+}\mathsf{True}],\\ & \quad\alpha.\mathsf{assert}:{+}\{\alpha{\equiv}\mathsf{True}\}\alpha\,[\vec{P}]\\ & \quad\alpha.\mathsf{nassert}:{+}\{\alpha{\equiv}\mathsf{False}\}\alpha]\end{array}$$

$\mathsf{True}{\Rightarrow}\vec{P}$ is identical to *Bool*, except that $\mathsf{True}{\Rightarrow}\vec{P}$ has the postcondition $\vec{P}$ on assert. The symmetric type $\mathsf{False}{\Rightarrow}\vec{P}$ places the postcondition on nassert rather than assert. Note that it is always safe to ignore postconditions, and thus $\mathsf{True}{\Rightarrow}(\vec{P},\vec{Q})$ is a subtype of $\mathsf{True}{\Rightarrow}\vec{P}$, which is a subtype of $\mathsf{True}{\Rightarrow}(\,\cdot\,)$, which is identical to *Bool*. Conversely, one may add postconditions in a supertype if those postconditions are proven to hold in the current environment: $\vec{P}\vdash \mathit{Bool}\leqslant(\mathsf{True}{\Rightarrow}\vec{P})$.

The typing rule S-CUT allows access to the postcondition of assert without actually calling it. One can derive the following variant of modus ponens:

$$\frac{E\vdash\alpha.f:{+}\mathsf{True}{\Rightarrow}\vec{P}\quad E\vdash\alpha.f:{+}\mathsf{True}\quad E\vdash *:\alpha}{E\vdash\vec{P}}$$

Monotone booleans with constraints. Our examples have many monotone boolean fields that are initially false, and remain true once they are set. We define the following derived form for constraints.

$$\begin{array}{rcl}\alpha.f:{\ominus}\exists(\vec{\tau})\,\mathsf{True}{\Rightarrow}\vec{P} & \triangleq & \alpha.f:{-}\exists(\vec{\tau})\,\mathsf{True}\,[\vec{P}]\\ \alpha.f:{\oplus}\exists(\vec{\tau})\,\mathsf{True}{\Rightarrow}\vec{P} & \triangleq & \alpha.f:{+}\exists(\vec{\tau})\,\mathsf{True}{\Rightarrow}(\vec{P},\,\alpha.f:{+}\mathsf{True})\end{array}$$

Internal types often have both constraints, abbreviated $\alpha.f:{\ominus\oplus}\mathsf{True}{\Rightarrow}\vec{P}$. If f returns false there are no guarantees on its future value, but if f returns true, then will henceforth always return true. Only the value true may be written to f, and it must be written in an environment E where $E\vdash\vec{P}$ in order to satisfy the postcondition of $\mathsf{True}\,[\vec{P}]$. Conversely, once a reader reads true on f, they may assume both that $\vec{P}$ hold and that f will always return true in future.

The soundness of the derived rule for conditionals follows from the type assigned to the visitor v in the encoding:

$$\begin{array}{l}\varsigma(\beta)\,[\beta.\mathsf{result}:{-}A,\\ \quad\beta.\mathsf{done}:{\ominus\oplus}\mathsf{True}{\Rightarrow}(\beta.\mathsf{result}:{+}\mathsf{Top}),\\ \quad\beta.\mathsf{istrue}:{+}\{\alpha{\equiv}\mathsf{True}\}\beta\,[\beta.\mathsf{done}:{+}\mathsf{True}],\\ \quad\beta.\mathsf{isfalse}:{+}\{\alpha{\equiv}\mathsf{False}\}\beta\,[\beta.\mathsf{done}:{+}\mathsf{True}]\,]\end{array}$$

Options. Option types can be defined as follows:

$$\begin{array}{rcl}\mathsf{none} & \triangleq & [\mathsf{isSet}=\mathsf{false},\,\mathsf{get}=\mathsf{fail}]\\ \mathsf{some} & \triangleq & \lambda(y)\,[\mathsf{isSet}=\mathsf{true},\,\mathsf{get}=y]\end{array}$$

We then have $\vdash\mathsf{none}:\mathit{None}$ and $\vdash\mathsf{some}:A\to\mathit{Some}(A)$, where

$$\begin{array}{rcl}\mathit{None} & \triangleq & \varsigma(\alpha)\,[\alpha.\mathsf{isSet}:{\boxplus}\mathsf{False},\,\alpha.\mathsf{get}:{+}\forall(\beta)\{\mathsf{True}{\equiv}\mathsf{False}\}\beta]\\ \mathit{Some}(A) & \triangleq & \varsigma(\alpha)\,[\alpha.\mathsf{isSet}:{\boxplus}\mathsf{True},\,\alpha.\mathsf{get}:{+}A]\end{array}$$

with the common supertype

$$\mathit{Option}(A) \triangleq \varsigma(\alpha)[\alpha.\mathsf{isSet}:\boxplus\mathit{Bool},\ \alpha.\mathsf{get}:+\{\alpha.\mathsf{isSet}:+\mathsf{True}\}A].$$

References. Our type system allows us to define uninitialized references, as in C, rather than initialized ones, as in Java/C#, yet accrue the benefits of Java-style initialization. Indeed, the guarantees of the type system are even stronger than those of Java/C# (and equivalent to those of ML): not only is memory initialized before it is accessed, but it is always initialized with something other than null. The implementation is as follows.

$$\begin{array}{l}\mathsf{ref} \triangleq [\mathsf{contents}=\mathsf{fail},\ \ \mathsf{isSet}=\mathsf{false},\\ \qquad \mathsf{get}=\varsigma(x)x.\mathsf{isSet}.\mathsf{assert};x.\mathsf{contents},\\ \qquad \mathsf{set}=\varsigma(x)\lambda(y)x.\mathsf{contents}\Leftarrow y;x.\mathsf{isSet}\Leftarrow\mathsf{true};x]\end{array}$$

For any A, ref has the following external and internal types.

$$\begin{array}{ll}\mathit{Ref}(A) \triangleq \varsigma(\alpha)[& \mathit{RefImpl}(A) \triangleq \varsigma(\alpha)[\\ \quad \alpha.\mathsf{isSet}:\oplus\mathit{Bool}, & \quad \alpha.\mathsf{contents}:-A,\\ \quad \alpha.\mathsf{get}:+\{\alpha.\mathsf{isSet}:+\mathsf{True}\}A, & \quad \alpha.\mathsf{isSet}:\ominus\oplus\mathsf{True}\Rightarrow(\alpha.\mathsf{contents}:+A),\\ \quad \alpha.\mathsf{set}:+A\to(\alpha[\alpha.\mathsf{isSet}:+\mathsf{True}])] & \quad \alpha.\mathsf{get}:+\{\alpha.\mathsf{isSet}:+\mathsf{True}\}A,\\ & \quad \alpha.\mathsf{set}:+A\to\alpha[\alpha.\mathsf{isSet}:+\mathsf{True}]]\end{array}$$

Adding an "unset" method to *Ref* would make it unimplementable, since isSet is monotone. The monotonicity property imposes the postcondition $\alpha.\mathsf{isSet}:+\mathsf{True}$, which leads to an absurdity in a program that allows isSet to be updated to false, which has type False.

4 Deserializing graphs

We consider a deserialization algorithm that builds a potentially cyclic graph from a textual description. This typifies the treatment of two important categories of algorithms: graph algorithms based on graph traversals and the construction of cyclic data structures. State of the art typing systems, such as (Qi and Myers 2009), are not able to validate this algorithm (Qi 2008).

Specifically, we treat the deserialization algorithm used in languages such as Java (Sun 2005), simplified slightly for presentation. A deserialization algorithm is (partially) correct if any graph that it returns is free of dangling pointers. Correctness is important since it allows for the returned graph to be traversed without checks for `null` pointers or `NullPointerExceptions`.

The correctness of the algorithm, when established manually, follows by induction on the spanning tree constructed by a depth-first traversal. The shape of the graph and its spanning tree are not known at compile time. Nonetheless, our type system is able to validate the algorithm. The proof constructed by the type system is an inductive argument using a static approximation of the spanning tree used in manual proofs. The key features that enable typing this example are existential quantification over preconditions, forward declarations using T-LET, and modus ponens using T-CUT.

The details may be found in Appendix C. A simpler extended example, showing workflow dependencies, is exhibited in Appendix B.

5 Related work

The related work falls into two categories, namely (a) session types for process languages, and (b) typestates in object-oriented languages.

Session types. Takeuchi et al. (1994) and Honda et al. (1998) identify session types — as we understand them today — in the context of the pi-calculus. See Honda et al. (2008) for a conceptual recent introduction. Gay and Hole (1999) initiate the study of subtyping for session types. Dezani-Ciancaglini et al. (2007) adds bounded polymorphism and typecase to the language of session types. Planul et al. (2009) adds concurrency to the vocabulary of session types. Igarashi and Kobayashi (2004) develop a relevant foundational technique by describing a general framework to permit a uniform way of describing and analysing a variety of type systems for the pi calculus. From a more foundational viewpoint, Honda et al. (2008) formalizes and studies multi-party protocols while still maintaining linearity restrictions on the connection implementing the portion of the session between any two given participants.

In terms of design and implementation efforts, Vasconcelos et al. (2006) study session types for a multi-threaded functional language. Gay et al. (2010) unify communication channels and their session types into distributed object-oriented programming yielding a form of typestates. Hu et al. (2008) integrate refined session types for conversations on Sockets into Java. Static validation ensures that each end of a socket connection conforms to a locally declared protocol. At runtime, before a connection is established, dynamic checks ensure that both ends of a socket connection have agreed to implement compatible protocols. The Singularity operating system (Fähndrich et al. 2006) uses session-types for first-class channels in dynamic communication networks. In Singularity, data as messages is exchanged over linear channels that obey contracts described as session-types. These contracts are validated statically. Furthermore, the linearity constraints facilitate the transmission of messages by reference instead of copying since the ownership is transferred along with the send.

However, in all of this work, copyable channels are stateless. Our paper motivates future investigations into a more liberal regime of copyable channels, one of whose ends is copyable, and whose session type satisfies a monotonicity restriction.

Typestates. Strom (1983) and Strom and Yemini (1986) seem to be among the earliest to define a notion of typestate. These early papers view typestate checking as a dataflow analysis to ensure that the data in a program follows the specification laid out in the typestate. In later papers, Strom and Yellin (1993) address the initialization problem using conditional assertions on typestates in the context of a backward dataflow analysis.

Chambers (1993) develops these ideas for object oriented programming as predicate classes. This paper views predicates as temporal properties that capture the changes in the behavior modes of the object and provides a design and implementation in the Cecil language. Motivated by the desire to capture the evolution of objects over time in object-oriented databases, Gottlob et al. (1996) add role hierarchies to object-oriented languages, using the current roles associated with an object as an abstract representation of its state. The role nomenclature reappears in this context in the more recent work

of Kuncak et al. (2002) where roles are used to represent precise and legal aliasing relationships that involve an object.

The Fickle project (Drossopoulou et al. 2002) considers typestates in the form of dynamic reclassification of the class membership of object references. In addition to a static type system, type-preserving translations into Java have also been investigated in this research (Ancona et al. 2007).

DeLine and Fähndrich (2001) have developed Vault, a variant of C that enables the specification and static validation of API usage rules. The same authors have also extended and elaborated these ideas in an object-oriented paradigm (DeLine and Fähndrich 2004).

Gay et al. (2008) view dynamic interfaces for objects from the standpoint of session types and consider object protocols for linear objects without aliasing. Xu (2001) views typestate checking as a flow-sensitive analysis and applies to safety-checking in scenarios where untrusted machine code is loaded into a trusted host system. In our paper, we use a rich language of types to perform some flow-sensitive analysis in the context of traditional type systems.

A common theme in the research on general typestates is that aliasing and its treatment by variants of linearity and ownership is one of the key technical issues that appears in the static program analysis for typestates, e.g., (Lam et al. 2005; Field et al. 2005; Fink et al. 2008; Xu 2001; Bierhoff and Aldrich 2007; Fähndrich and DeLine 2002) is an incomplete list of a variety of ways to attack this difficult problem. In addition to more traditional conservative ("may") approximations to alias information, this line of research reveals that typestate analysis also benefits from "must"-alias information; e.g., two references do point to the same object. This observation is reflected in our type system in the association of object references to type names.

The most immediate inspiration for our own work is Fähndrich and Leino (2003)'s study of monotone typestates. Our techniques are however rather different from this paper, as reflected in the assignment rule. They permit assignment only if the value being assigned has a fully formed immutable typestate, so that no future evolution is possible for it. In contrast, our assignment rule (for fields and updatable methods) is traditional and only constrains the assigned value to be a subtype of the declared type of the field/method. This paper also suggests that the expressiveness of the typesystem be "measured" via its ability to handle cyclic data structures. In contrast to their treatment by a mixture of dynamic typestate tests and two-pass solutions, we are able to statically validate the traditional single-pass algorithms. On the other hand, we do not address the inheritance issues that are treated in Fähndrich and Leino (2003).

(Fähndrich and Xia 2007) uses delayed types and an associated block-structured construct to facilitate the static validation of initialization of (perhaps circular) data structures. They leave open the problem of integrating phased initialization protocols into the framework.

This problem is solved to large extent in a recent paper by Xin Qi and Andrew C. Myers (Qi and Myers 2009). The typesystem of this paper specifies conditions on when a field gets initialized in terms of when when other reachable field(s) are initialized.

This idea is illustrated by returning to the example from the introduction:

$$\mathsf{let}\ x = [\mathsf{f} = \mathsf{fail}];\ \mathsf{let}\ y = [\mathsf{fst} = \mathsf{fail}, \mathsf{snd} = \mathsf{fail}];\ \mathsf{let}\ v = [\,];$$
$$x.\mathsf{f} \Leftarrow y;\ x.\mathsf{f}.\mathsf{fst} \Leftarrow v;\ x.\mathsf{f}.\mathsf{snd} \Leftarrow x$$

In this example, initialization of x is complete when initialiation of its f field is. In turn, initialization of the f field depends on y. The initialization of the snd field of y depends on x again. The typesystem recognizes and discharges this cyclic dependency between x and the snd field of y. In general, (Qi and Myers 2009) provide an elegant proof rule to eliminate strongly connected components of dependencies. Conceptually, this step is motivated by a bisimulation argument.

However, this elegant idea cannot handle the typing of the standard deserialization algorithm that we have described earlier in this paper. The condition that a node is properly initialized when the root of the graph is properly initialized is a conditional dependency on a potentially unreachable root node, so not expressible in the system of (Qi and Myers 2009) that only can usefully address conditional dependencies on nodes reachable in the object reference graph.

6 Conclusions and Future work

This paper contributes to the research program exploring expressive specifications to facilitate static validations of complex behavioral and runtime properties. Our main contribution is a complete formal treatment of monotone typestates for the object calculus that avoids the aliasing restrictions that have encumbered other typestate proposals.

While the basic ingredients of our type system are already found in the literature, their application to monotone typestates advances the state of the art. We have shown that our methods address the twin challenges of phased initialization protocols and (the creation and traversal of) cyclic data structures. In particular, our treatment of cyclic data structures are for the standard one-pass algorithms, and our fully static validation does not require any dynamic typestate checks.

We have shown that we can code Full System $F_{<:}$ into our type system, and hence (Pierce 1991) subtyping is undecidable. We expect that the same techniques for algorithmic subtyping adopted by languages such as Java would apply to this system, that is: 1. Require all variables to be explicitly typed by the programmer, thus turning many instances of type inference into typechecking. 2. Require the programmer to supply the subtyping relation explicitly, by means of "extends" and "implements" annotations, thus turning the problem of computing a subtype relation into one of checking that a programmer-provided relation is a subtype relation. We leave a precise development of this idea to future work.

In this paper, we consider only a small core type system that suffices to illustrate the basic ideas and address the examples. A fuller treatment in the context of a realistic programming language that includes recursive types, value type polymorphism and scaled up to include classes and inheritance awaits future work.

The most exciting prospects are in concurrency that we do not explicitly consider in this paper. It seems highly plausible that our monotonicity properties are compatible with concurrent computation modulo the addition of suitable atomicity assumptions. We leave this development to future work.

Acknowledgements. We thank Qi and Myers for a patient and detailed discussion of their typesystem; of course, we take responsibility for any errors in our discussion of their work. We also heartily thank the anonymous referees of prior versions of this paper.

Bibliography

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. A provenly correct Translation of Fickle into Java. *ACM TOPLAS*, 2, 2007.

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320. ACM, 2007.

C. Chambers. Predicate classes. In *ECOOP*, volume 707 of *LNCS*, pages 268–296. Springer, 1993.

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, volume 3086 of *LNCS*, pages 465–490. Springer, 2004.

M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded Session Types for Object-Oriented Languages. In *FMCO'06*, volume 4709 of *LNCS*, pages 207–245. Springer-Verlag, 2007.

S. Drossopoulou, F. Damiani, D. Dezani-Ciancaglini, and P. Giannini. More Dynamic Object Re-classification: FickleII. *ACM TOPLAS*, pages 153–191, March 2002.

M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO)*, 2003.

M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM, 2007.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190. ACM, 2006.

J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program.*, 58(1-2):57–82, 2005.

S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

S. J. Gay and M. Hole. Types and subtypes for client-server interactions. In *ESOP*, volume 1576 of *LNCS*, pages 74–90. Springer, 1999.

S. J. Gay, A. Ravara, and V. T. Vasconcelos. Dynamic interfaces. http://www.dcs.gla.ac.uk/~simon/publications/DynamicInterfaces.pdf, July 2008.

S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, 2010. To appear.

G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP*, volume 5142, 2008. To appear.

G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007. doi: http://doi.acm.org/10.1145/1243418.1243424.

A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

V. Kuncak, P. Lam, and M. C. Rinard. Role analysis. In *POPL*, pages 17–32, 2002.

P. Lam, V. Kuncak, and M. C. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI*, volume 3385 of *LNCS*, pages 430–447. Springer, 2005.

G. Lindstrom. Functional programming and the logical variable. In *POPL*, pages 266–280, 1985.

D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Log.*, 6(4):749–783, 2005.

B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.

J. Planul, R. Corin, and C. Fournet. Secure enforcement for global process specifications. In M. Bravetti and G. Zavattaro, editors, *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2009.

X. Qi. Private communication, December 2008.

X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65, 2009.

R. E. Strom. Mechanisms for compile-time enforcement of security. In *POPL*, pages 276–284. ACM, 1983.

R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Software Eng.*, 19(5):478–485, 1993.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

Java Object Serialization Specification. Sun Microsystems, 2005. http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html.

K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1994.

Z. Xu. *Safety-Checking of Machine Code*. PhD thesis, University of Wisconsin-Madison, 2001.

A Proof sketch for the weak safety theorem

Our proof of weak safety follows the standard approach that "safety is preservation plus progress". Both results require that configurations be well formed. This, in turn, requires that we choose an environment that is both well formed and compatible with the store.

$$\frac{}{\cdot \vdash \diamond}\ \text{(E-EMPTY)} \qquad \frac{E \vdash \diamond \quad \tau \notin dom(E)}{E, \tau \vdash \diamond}\ \text{(E-TYPE)} \qquad \frac{E \vdash \diamond \quad \tau \in dom(E) \quad v \notin dom(E)}{E, v : \tau \vdash \diamond}\ \text{(E-ID)}$$

$$\frac{E \vdash \diamond \quad fid(P) \subseteq dom(E) \quad \forall((\phi.\ell : -F) \in (E,P)) \quad \forall((\phi.\ell : \chi G) \in (E,P))\ \ E \vdash F \leqslant G}{E, P \vdash \diamond}\ \text{(E-CON)}$$

$$\frac{\forall((\phi.\ell : -\forall(\vec{\alpha})A) \in E)\ \exists(i)\ (p_i : \phi) \in E \quad \forall((\phi.\ell : +\forall(\vec{\alpha})A) \in E)\ \exists(i)\ (p_i : \phi) \in E\ \ \exists((\ell = \varsigma(x)a) \in o_i)\ \ E, \vec{\alpha}, x : \phi \vdash a : A}{E \vdash p_1 = o_1, \ldots, p_n = o_n}\ \text{(T-STORE)}$$

$$\frac{E \vdash \diamond \quad E \vdash S \quad E \vdash a : A}{\vdash S /\!\!/ a : A}\ \text{(T-CONFIG)}$$

Note that T-CONFIG allows one to choose E to satisfy the requirements.

Preservation does not usually hold in the standard form when non-monotonic typestates are involved. So, the standard statement of preservation for our type system is actually a witness to its monotonicity.

Proposition (Preservation). *If* $\vdash S /\!\!/ a : A$ *and* $S /\!\!/ a \to S' /\!\!/ a'$ *then* $\vdash S' /\!\!/ a' : A$. □

The proof of preservation follows standard methods (substitution lemma, induction over the definitions of evaluation and typing). The requirement is that at each step of evaluation, we be able to find an environment that will type the resulting configuration. The environment that we choose will evolve as evaluation proceeds. In particular note that T-UPD adds new capabilities to the consequent using a postcondition. After an update, therefore, the choice of environment may change in the consequent. Thus, the principal types of all the objects involved becomes more refined (lower in the subtype order) as programs evolve.

Proposition (Progress). *If* $\vdash S /\!\!/ a : V$ *then either* a *is a name or* $S /\!\!/ a \to S' /\!\!/ a'$, *for some* S', a'. □

Progress is possible only if the top level environment discharges all the required preconditions. Thus, progress fails if one replaces V (which does not permit preconditions) by A (which permits preconditions).

B Workflow Dependencies

Boolean fields can be used in pre- and post-conditions to force method calls to occur in a certain order. For example, the following external type enforces that f_1 and f_2 must be called before g.

$$\begin{array}{l}\varsigma(\alpha)\,[\alpha.\mathsf{f}_1 :+ \alpha\,[\alpha.\mathsf{b}_1 :+ \mathsf{True}]\,,\ \alpha.\mathsf{b}_1 :+ \mathit{Bool},\\ \qquad\qquad \alpha.\mathsf{f}_2 :+ \alpha\,[\alpha.\mathsf{b}_2 :+ \mathsf{True}]\,,\ \alpha.\mathsf{b}_2 :+ \mathit{Bool},\\ \qquad\qquad \alpha.\mathsf{g} :+ \{\alpha.\mathsf{b}_1 :+ \mathsf{True},\ \alpha.\mathsf{b}_2 :+ \mathsf{True}\}\alpha]\end{array}$$

After the call to f_1 terminates, a client can be sure that any future read of b_1 will return true (or more accurately, a value of type True, of which there is only one). Thus, the call to f_1 *improves* the type of b_1 by lowering it to a subtype.

Generalizing this idea to represent arbitrary workflows (as compile-time DAGs), we define the factory WF_n which builds task objects that depend on n other task objects. The build method expects a vector $\vec{x}$ of n other tasks and a function f to be run as the body of the generated task. The monotone boolean field done in the returned object is used to track the completion of the task. (We elide several notations involving vectors.)

$$\begin{array}{l}\mathsf{WF}_n \triangleq [\mathsf{build} = \lambda(\vec{x}, f)\,[\mathsf{done} = \mathsf{false},\\ \qquad\qquad\qquad \mathsf{run} = \varsigma(y)\vec{x}.\mathsf{done}.\mathsf{assert};\ \mathsf{let}\ z = f.\mathsf{run};\ y.\mathsf{done} \Leftarrow \mathsf{true};\ z]]\end{array}$$

The build method of WF_n can be assigned the following external type (where $n = |\vec{\alpha}|$).

$$\begin{array}{l}\forall(\vec{\alpha}, \beta, \gamma)\{\vec{\alpha}.\mathsf{done} :+ \mathit{Bool}, \beta.\mathsf{run} :+ \gamma\}(\vec{\alpha} \times \beta) \to \varsigma(\alpha)\,[P_n(\vec{\alpha}, \alpha, \gamma)]\\ \qquad P_n(\vec{\alpha}, \alpha, \gamma) \triangleq \alpha.\mathsf{done} :\oplus \mathsf{True} \Rightarrow (\vec{\alpha}.\mathsf{done} :+ \mathsf{True}),\\ \qquad\qquad\qquad \alpha.\mathsf{run} :+ \{\vec{\alpha}.\mathsf{done} :+ \mathsf{True}\}\gamma[\alpha.\mathsf{done} :+ \mathsf{True}]\end{array}$$

The type parameters $\vec{\alpha}$ are used in the precondition of run and in the conditional boolean type of done, which states that whenever $\alpha.\mathsf{done}$ is true, it must be that $\vec{\alpha}.\mathsf{done}$ are true.

As an example, consider a workflow with four tasks order as follows: $1 \to 2 \to 4$, $1 \to 3 \to 4$. An object storing the tasks can be created as follows.

$$\begin{array}{l}\mathsf{let}\, f_1 = \ldots;\ \mathsf{let}\, f_2 = \ldots;\ \mathsf{let}\, f_3 = \ldots;\ \mathsf{let}\, f_4 = \ldots;\\ \mathsf{let}\, x_1 = \mathsf{WF}_0.\mathsf{build}\,((), f_1);\quad \mathsf{let}\, x_2 = \mathsf{WF}_1.\mathsf{build}\,((x_1), f_2);\\ \mathsf{let}\, x_3 = \mathsf{WF}_1.\mathsf{build}\,((x_1), f_3);\ \ \mathsf{let}\, x_4 = \mathsf{WF}_2.\mathsf{build}\,((x_2, x_3), f_4);\\ \mathsf{let}\, t = [\mathsf{task}_1 = x_1, \mathsf{task}_2 = x_2, \mathsf{task}_3 = x_3, \mathsf{task}_4 = x_4];\ a\end{array} \qquad (*)$$

The environment used to type the residual term a tracks the dependencies:

$$\begin{array}{l}E = \vec{\gamma}, f : \vec{\gamma}, \vec{\alpha}, \vec{x} : \vec{\alpha}, \delta, t : \delta,\\ \quad \delta.\mathsf{task}_1 :+ \alpha_1,\ \delta.\mathsf{task}_2 :+ \alpha_2,\ P_0((), \alpha_1, \gamma_1),\quad P_1((\alpha_1), \alpha_2, \gamma_2),\\ \quad \delta.\mathsf{task}_3 :+ \alpha_3,\ \delta.\mathsf{task}_4 :+ \alpha_4,\ P_1((\alpha_1), \alpha_3, \gamma_3),\ P_2((\alpha_2, \alpha_3), \alpha_4, \gamma_4)\end{array}$$

When task i is accessed at the corresponding type α_i, the preconditions on its run method refer precisely to the type names of the fields of the other tasks. This allows the postcondition of one task (or testing of its done field) to be used to satisfy a precondition of another task.

To see how typing works, suppose that the term a in the code above is "$t.\mathsf{task}_2.\mathsf{run}$". We have that $t.\mathsf{task}_2 : \alpha_2$. Using P_1 and subsumption to establish the premise, we can use T-SEL to deduce

$$\frac{E \vdash t.\mathsf{task}_2 : \alpha_2\,[\alpha_2.\mathsf{run} : +\{\alpha_1.\mathsf{done} : +\mathsf{True}\}\gamma_2\,[\alpha_2.\mathsf{done} : +\mathsf{True}]\,]}{E \vdash t.\mathsf{task}_2.\mathsf{run} : \{\alpha_1.\mathsf{done} : +\mathsf{True}\}\gamma_2\,[\alpha_2.\mathsf{done} : +\mathsf{True}]}.$$

The term (*), in which a occurs, is a well-typed term, but it is not a well-typed *program* since the precondition on $t.\mathsf{task}_2.\mathsf{run}$ has not been discharged.

With a set to any of the following, however, the term is typeable as a program.

$E \vdash (t.\mathsf{task}_1.\mathsf{run};\ t.\mathsf{task}_2.\mathsf{run};\ t.\mathsf{task}_3.\mathsf{run};\ t.\mathsf{task}_4.\mathsf{run}) : \gamma_4$

$E \vdash (\mathsf{if}\ (t.\mathsf{task}_1.\mathsf{done})\ \mathsf{then}\ (t.\mathsf{task}_2.\mathsf{run})\ \mathsf{else}\ (t.\mathsf{task}_1.\mathsf{run};\ t.\mathsf{task}_2.\mathsf{run})) : \gamma_2$

$E \vdash (\mathsf{if}\ (t.\mathsf{task}_4.\mathsf{done})\ \mathsf{then}\ (t.\mathsf{task}_1.\mathsf{done}.\mathsf{assert})\ \mathsf{else}\ \mathsf{true}) : \mathit{Bool}$

The last of these requires S-CUT to deduce that task_1 must be completed whenever task_4 is complete.

C Deserializing graphs

To simplify the presentation, we assume that each node has exactly two children. The smallest such binary graph has a single node, which is its own left and right child. For concreteness, let the input be a description of the graph using node names encoded as strings. (We do not discuss error handling for poorly formed descriptions.)

$$\begin{aligned}\mathit{Desc} \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{getRoot} : +\mathit{String},\\ \alpha.\mathsf{getLeft} : +\mathit{String} \to \mathit{String},\\ \alpha.\mathsf{getRight} : +\mathit{String} \to \mathit{String}]\end{aligned}$$

If d describes the smallest graph and $d.\mathsf{getRoot}$ returns `"bob"`, then $d.\mathsf{getLeft}(\texttt{"bob"})$ and $d.\mathsf{getRight}(\texttt{"bob"})$ should also return `"bob"`.

The deserialization problem, then, is to define a function of type $\mathit{Desc} \to \mathit{Node}$, where *Node* is defined as follows.

$$\mathit{Node} \triangleq \mu(\theta)\varsigma(\alpha)\,[\alpha.\mathsf{getName} : +\mathit{String},\ \alpha.\mathsf{getLeft} : +\theta,\ \alpha.\mathsf{getRight} : +\theta]$$

Using simple sugar for recursive types, we write this as follows.

$$\mathit{Node} \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{getName} : +\mathit{String},\ \alpha.\mathsf{getLeft} : +\mathit{Node},\ \alpha.\mathsf{getRight} : +\mathit{Node}]$$

Since our language does not have `null`, the type *Node* ensures that the returned graph is fully initialized: getLeft and getRight may be called to arbitrary depth in the graph, and the returned node is always guaranteed to be properly initialized. Null pointer exceptions cannot happen.

For clarity of presentation, we express the base case of the inductive argument using a separate container object for the root node of the graph. The container object has type *Root*, defined as follows.

$$\mathit{Root} \triangleq \varsigma(\rho)\,[\rho.\mathsf{get} : +\mathit{Node}]$$

Our goal, then, is to define an object GRAPHFACTORY and to prove that it has type

$$\mathit{GraphFactory} \triangleq \varsigma(\phi)\,[\phi.\mathsf{buildRoot}:+\mathit{Desc} \to \mathit{Root}]\,.$$

Given a description d, one can then retrieve the root node of the deserialized graph by calling GRAPHFACTORY.buildRoot(d).get.

The implementation of buildRoot requires an auxiliary function buildNode to create the nodes of the graph. In turn, buildNode uses a lookup table to record nodes which are already constructed or are in the process of construction. We elide the implementation of TABLEFACTORY which produces tables. The table maps *String*s to nodes. For nodes, we use the internal type $\mathit{NodeImpl}(\rho)$, defined below. Here ρ is the typename of the root container to which the node belongs. This dependence from node to root allows the type of the node to conditionally refer to montone boolean fields in the root container object. We leverage this to give nodes a type indicating that whenever the root is done, so must the node be done.

Throughout the example, we use the following naming conventions:

- f is the factory with typename ϕ bounded by *GraphFactory* and *GraphFactoryImpl*,
- d is the description of type *Desc*,
- s is a variable of type *String*,
- r is the root with typename ρ bounded by *Root* and *RootImpl*,
- t is the table of type $\mathit{Table}(\rho)$,
- n is a variable with typename α bounded by *Node* and $\mathit{NodeImpl}(\rho)$,
- b is a variable of type *Bool*, and
- x is a self variable of type *Root* or *Node*.

We define GRAPHFACTORY to have the following implementation and internal type, where $\mathit{Table}(\rho)$ is the type of maps from *String* to $\mathit{NodeImpl}(\rho)$. (The ellipses will be filled in below.)

$$\begin{aligned}
&\mathsf{GRAPHFACTORY} = [\mathsf{buildRoot} = \varsigma(f)\lambda(d)\cdots, \mathsf{buildNode} = \varsigma(f)\lambda(d,s,t)\cdots]\\
&\mathit{GraphFactoryImpl} \triangleq \varsigma(\phi)\,[\phi.\mathsf{buildRoot}:+\mathit{Desc} \to \cdots\\
&\qquad\qquad \phi.\mathsf{buildNode}:+\forall(\rho)(\mathit{Desc} \times \mathit{String} \times \mathit{Table}(\rho)) \to \cdots]\\
&\qquad \mathit{Table}(\rho) \triangleq \varsigma(\alpha)\,[\alpha.\mathsf{contains}:+\mathit{String} \to \mathit{Bool},\\
&\qquad\qquad \alpha.\mathsf{get}:+\mathit{String} \to \mathit{NodeImpl}(\rho),\\
&\qquad\qquad \alpha.\mathsf{add}:+\mathit{String} \times \mathit{NodeImpl}(\rho) \to \alpha]
\end{aligned}$$

The code for buildRoot is as follows. The code creates root container objects, which have two fields and one method: The field node holds the root node; it is initially set to fail and is only set again after the entire graph is initialized. The field done is initially false; it is set to true after node is set. The method get is a simple accessor for field node. In the code, f is the self parameter for the factory and d is the formal parameter for the input description.

```
buildRoot = ς(f)λ(d)
  let r = [node = fail, done = false, get = ς(x)x.node];
  let t = TABLEFACTORY.buildTable;
  let n = f.buildNode(d, d.getRoot, t);
  r.node ⇐ n;
  r.done ⇐ true;
  r
```

After initializing the root container r and the lookup table t, buildNode does the heavy work of constructing the graph, storing the root node in n. Subsequently, the appropriate fields of r are set and r is returned.

The code can be assigned the following type, which refers to the type $\mathit{NodeImpl}(\rho)$, defined later. (The constraint $\rho \leq \mathit{RootImpl}$ uses the conventions for notating bounds with existential binders and self types, described at the beginning of Section 3.)

$$\begin{array}{l}
\phi.\mathsf{buildRoot}:+\mathit{Desc} \to \varsigma(\rho)\,[\rho \leq \mathit{RootImpl},\ \rho.\mathsf{done}:+\mathsf{True}] \\
\mathit{RootImpl} \triangleq \exists(\alpha)\varsigma(\rho)\,[\\
\quad \rho.\mathsf{node}:-\alpha, \\
\quad \rho.\mathsf{done}:\ominus\oplus\mathsf{True} \Rightarrow (\alpha \leq \mathit{NodeImpl}(\rho),\ \alpha.\mathsf{done}:+\mathsf{True}), \\
\quad \rho.\mathsf{get}:+\{\alpha.\mathsf{done}:+\mathsf{True}\}\alpha \\
]
\end{array}$$

The postcondition of buildNode, defined below, exactly matches the condition on done. Given this, it is relatively straightforward to establish that buildRoot types as advertised above. Our goal, however, is to show that buildRoot has type $\mathit{Desc} \to \mathit{Root}$, where Root is defined to be $\varsigma(\rho)\,[\rho.\mathsf{get}:+\mathit{Node}]$. We can achieving this more abstract type via subsumption. $\mathit{NodeImpl}(\rho)$ is defined so that

$$\text{if } E \vdash \rho.\mathsf{done}:+\mathsf{True} \text{ then } E \vdash \mathit{NodeImpl}(\rho) \leqslant \mathit{Node}.$$

Using this fact, it follows that

$$\text{if } E \vdash \rho.\mathsf{done}:+\mathsf{True} \text{ and } E \vdash \rho \leqslant \mathit{RootImpl} \text{ then } E \vdash \rho \leqslant \mathit{Root}.$$

One can then use subsumption to assign buildRoot the desired type.

We now turn to the code for buildNode. The nodes being built have three methods and six fields: The methods are all simple accessors. The field name returns the name of the node as a string. The fields left and right return pointers to the left and right child nodes; these are initially set to fail. The fields oldL and oldR establish the birth order between this node and its left and right children—if oldL is false, then this node is created before its left child; these fields are set immutably when the object is first constructed. The field done is initially false; it is set to true after left and right are set.

The code follows. Again, f is the self parameter for the factory and d is the formal parameter for the input description. The formal parameter s provides the name of the node to be constructed (as a string). The formal parameter t provides the lookup table.

```
buildNode = ς(f)λ(d,s,t)
  let s_L = d.getLeft(s);   let b_L = s.eq(s_L) || t.contains(s_L);
  let s_R = d.getRight(s);  let b_R = s.eq(s_R) || t.contains(s_R);
  let n = [name = s,      getName = ς(x)x.name,
           left = fail,   getLeft = ς(x)x.left,     oldL = b_L,
           right = fail,  getRight = ς(x)x.right,   oldR = b_R,
           done = false];
  t.add(s,n);
  let n_L = if n.oldL then t.get(s_L) else f.buildNode(d,s_L,t);  n.left ⇐ n_L;
  let n_R = if n.oldR then t.get(s_R) else f.buildNode(d,s_R,t);  n.right ⇐ n_R;
  n.done ⇐ true;
  n
```

The code first establishes the birth order with respect to the child nodes. It then creates the node and adds it to the lookup table. The left and right children are then set. If oldL is true then this is a simple table lookup, otherwise there is a recursive call to buildNode to build the left child. Likewise for the right child. Finally the done field is set and the node is returned.

The code can be assigned the following type.

$$\begin{array}{l}\phi.\mathsf{buildNode}:+\forall(\rho)(\mathit{Desc}\times\mathit{String}\times\mathit{Table}(\rho))\to\\ \qquad\qquad\exists(\alpha)\{*:\rho,\ \rho.\mathsf{done}:+\mathsf{True}\Rightarrow(\alpha.\mathsf{done}:+\mathsf{True})\}\\ \qquad\qquad\quad\alpha\\ \qquad\qquad\quad[\alpha\leq\mathit{NodeImpl}(\rho),\ \alpha.\mathsf{done}:+\mathsf{True}]\end{array}$$

$$\begin{array}{l}\mathit{NodeImpl}(\rho)\triangleq\exists(\alpha_L,\alpha_R)\mathbf{\varsigma}(\alpha)[\\ \quad\alpha.\mathsf{name}:\boxplus\mathit{String},\ \alpha.\mathsf{getName}:+\mathit{String},\\ \quad\alpha.\mathsf{left}:-\alpha_L,\qquad\alpha.\mathsf{getLeft}:+\alpha_L,\qquad\alpha.\mathsf{oldL}:\boxplus\mathit{Bool}\\ \quad\alpha.\mathsf{right}:-\alpha_R,\qquad\alpha.\mathsf{getRight}:+\alpha_R,\qquad\alpha.\mathsf{oldR}:\boxplus\mathit{Bool}\\ \quad\alpha.\mathsf{done}:\ominus\oplus\mathsf{True}\Rightarrow(\\ \qquad\alpha.\mathsf{left}:+\alpha_L,\quad\alpha_L\leq\mathit{NodeImpl}(\rho),\ \alpha.\mathsf{oldL}:+\mathsf{False}\Rightarrow(\alpha_L.\mathsf{done}:+\mathsf{True}),\\ \qquad\alpha.\mathsf{right}:+\alpha_R,\ \alpha_R\leq\mathit{NodeImpl}(\rho),\ \alpha.\mathsf{oldR}:+\mathsf{False}\Rightarrow(\alpha_R.\mathsf{done}:+\mathsf{True})),\\ \quad\rho.\mathsf{done}:+\mathsf{True}\Rightarrow(\alpha.\mathsf{done}:+\mathsf{True})\\ ]\end{array}$$

The precondition of buildNode establishes that the root container exists ($*:\rho$) and that when the root container sets its done field, then each built node must have already set its own done field.

The postcondition of done states that, when set, the child pointers must both point to nodes. Further, it states that the order in which the done flags are set corresponds to birth order: if the left child is younger, then it is set first. If oldL false, then the node's left child is beneath it in the DFS order, i.e., it is a left child in the spanning tree constructed by the algorithm. Together, these ensure that when the root node is done, all reachable nodes have initialized children. Since a node sets done only after its children in the spanning tree, we deduce that if oldL is false, then the left child sets its done flag before the parent.

The type of the buildNode method illustrates the power of nesting the precondition the existential quantifier that is instantiated by the method—in this case, the precondition speaks about an yet to be generated node α that is the type of the return value.

In typing buildNode, the chief difficulty is in the recursive calls for the children. For concreteness, we discuss the recursive call for the left child:

$$\mathsf{if}\ n.\mathsf{oldL}\ \mathsf{then}\ t.\mathsf{get}(s_L)\ \mathsf{else}\ f.\mathsf{buildNode}(d,s_L,t).$$

To type this, we must show that $f.\mathsf{buildNode}$ satisfies its precondition:

$$E\vdash\rho.\mathsf{done}:+\mathsf{True}\Rightarrow(\alpha_L.\mathsf{done}:+\mathsf{True}).$$

The type system reasons as follows. Given that α is bounded by $\mathit{NodeImpl}(\rho)$, we can use the definition of $\mathit{NodeImpl}(\rho)$ and subtyping to conclude

$$E\ \vdash\ \alpha.\mathsf{done}:+\mathsf{True}\Rightarrow(\alpha.\mathsf{oldL}:+\mathsf{False}\Rightarrow(\alpha_L.\mathsf{done}:+\mathsf{True})).$$

We have $E \vdash * : \alpha$, witnessed by n, and therefore

$$E, \alpha.\mathsf{done} :+ \mathsf{True} \;\vdash\; \alpha.\mathsf{oldL} :+ \mathsf{False} \Rightarrow (\alpha_L.\mathsf{done} :+ \mathsf{True}).$$

We know that for some δ, $E \vdash \alpha.\mathsf{oldL} :+ \delta$ (since oldL is immutable) and that $E \vdash \delta \equiv \mathsf{False}$ (since we're in the else branch of a conditional); therefore, we may conclude

$$E, \alpha.\mathsf{done} :+ \mathsf{True} \;\vdash\; \alpha_L.\mathsf{done} :+ \mathsf{True}. \qquad (\dagger)$$

From the precondition of the outer call to buildNode we have

$$E \vdash * : \rho \qquad\qquad E \;\vdash\; \rho.\mathsf{done} :+ \mathsf{True} \Rightarrow (\alpha.\mathsf{done} :+ \mathsf{True})$$

from which we derive

$$E, \rho.\mathsf{done} :+ \mathsf{True} \;\vdash\; \alpha.\mathsf{done} :+ \mathsf{True}. \qquad (\ddagger)$$

Combining (†) and (‡) we have $E, \rho.\mathsf{done} :+ \mathsf{True} \;\vdash\; \alpha_L.\mathsf{done} :+ \mathsf{True}$ from which we derive $E \vdash \rho.\mathsf{done} :+ \mathsf{True} \Rightarrow (\alpha_L.\mathsf{done} :+ \mathsf{True})$ as required.