Chapter 9

Generative invariants

So far, we have presented SLS as a framework for presenting transition systems. This view focuses on synthetic transitions as a way of relating pairs of process states, either with one transition $(\Psi;\Delta) \rightsquigarrow (\Psi';\Delta')$ or with a series of transitions $(\Psi;\Delta) \rightsquigarrow^* (\Psi';\Delta')$. This chapter will focus on another view of concurrent SLS specifications as *grammars* for describing well-formed process states. This view was presented previously in the discussions of adequacy in Section 4.4.1 and in Section 6.3.

The grammar-like specifications that describe well-formed process states are called *generative signatures*, and generative signatures can be used to specify sets of process states, or *worlds*. By the analogy with grammars, we could also describe worlds as *languages* of process states recognized by the grammar. In our previous discussions of adequacy in Section 4.4.1 and in Section 6.3, the relevant world was a set of process states that we could put in bijective correspondence with the states of an abstract machine.

Generative signatures are a significant extension of context-free grammars, both because of the presence of dependent types and because of the presence of linear and persistent resources in SLS. However, we will not endeavor to study generative signatures in their own right in this chapter or this dissertation. Rather, we will use generative signatures for one very specific purpose: showing that, under some generative signature Σ_{Gen} that defines a world $\mathcal{W}$, whenever $(\Psi;\Delta) \in \mathcal{W}$ and $(\Psi;\Delta) \rightsquigarrow_\Sigma (\Psi';\Delta')$ it is always the case that $(\Psi';\Delta') \in \mathcal{W}$. (The signature Σ encodes the transition system we are studying.) In such a case, a world or language of well-formed process states is called a *generative invariant* of Σ.

Type preservation

Narrowing our focus even further, in this chapter our sole use of generative invariants will be describing well-formedness and well-typedness invariants of the sorts of substructural operational semantics specifications presented in Part II. When we want to prove language safety for a small-step SOS specification like $e \mapsto e'$ from Section 6.6.2 and the beginning of Chapter 6, we define a judgment $x_1{:}tp_1, \ldots, x_n{:}tp_n \vdash e : tp$. This *typing judgment* expresses that e has type tp if the expression variables $x_1, \ldots, x_n$ are respectively assumed to have the types $tp_1, \ldots, tp_n$. (Note that tp is an *object-level type* as described in Section 9.3, not an LF type τ from Chapter 4.)

Well-typedness invariants are important because they allow us to prove *language safety*, the

property (discussed way back in the introduction) that a language specification is completely free from undefined behavior. The standard "safety = progress + preservation" formulation of type safety is primarily a statement about invariants. We specify some property ("well-typed with type tp"), show that it is invariant under execution (preservation: "if $e \mapsto e'$ and e has type tp, then e' has type tp"), and show that any state with that property has well-defined behavior (progress: "if e has type tp, it steps or is a value").

The purpose of this chapter is to demonstrate that generative invariants are a solid methodology for describing invariants of SLS specifications, especially well-formedness and -typedness invariants of substructural operational semantics specifications like the ones presented in Part II. As we have already seen, well-formedness invariants are major part of adequacy theorems. In the next chapter, we will show that well-typedness invariants are sufficient for proving progress theorems, meaning that generative invariants can form the basis of progress-and-preservation-style safety theorems for programming languages specified in SLS. These two chapters support the third refinement of our central thesis:

> **Thesis (Part III):** *The SLS specification of the operational semantics of a programming language is a suitable basis for formal reasoning about properties of the specified language.*

Overview

In Section 9.1 we review how generative signatures define a world and show how the *regular worlds* that Schürmann implemented in Twelf [Sch00] fall out as a special case of the worlds described by generative signatures. After this, the core of this chapter plays the same game – describing a well-formedness or well-typedness property with a generative signature and proving that the property is a generative invariant – five times. In each step, we motivate and explain new concepts.

- In Section 9.2 we extend the well-formedness invariant for sequential ordered abstract machines described in Section 6.3 to parallel ordered abstract machines with failure, setting up the basic pattern.
- In Section 9.3 we switch from specifying well-*formed* process states to specifying well-*typed* process states. This is not a large technical shift, but conceptually it is an important step from thinking about adequacy properties to thinking about preservation theorems.
- In Section 9.4 we describe how generative invariants can be established for the sorts of stateful signatures considered in Section 6.5. This specification introduces the *promise-then-fulfill* pattern and also requires us to consider *unique index* properties of specifications (Section 9.4.2).
- In Section 9.5 we consider invariants for specifications in the image of the destination-adding transformation from Chapter 7. This formalization, which is in essence a SLS encoding of Cervesato and Sans's type system from [CS13], also motivates the introduction of *unique index sets* to state unique index properties more concisely.
- In Section 9.6 we consider the peculiar case of first-class continuations, which require us to use persistent continuation frames as described in Section 7.2.4. Despite the superficial

similarities between the SSOS semantics for first-class continuations and the other SSOS semantics considered in this dissertation, first-class continuations fundamentally change the control structure, and this is reflected in a fundamental change to the necessary generative invariants.

We conclude in Section 9.7 with a brief discussion of the mechanization of generative invariants, though this is primarily left for future work. In general, this chapter aims to be the first word in the use of generative invariants, but it is by no means the last.

9.1 Worlds

Worlds are nothing more or less than sets of stable process states $(\Psi; \Delta)$ as summarized in Appendix A. In this chapter, we will specify worlds with the combination of two artifacts: an initial process state and a generative signature.

Definition 9.1. *A* generative signature *is a SLS signature where the ordered, mobile, and persistent atomic propositions can be separated into two sets – the* terminals *and the* nonterminals*. Synthetic transitions enabled by a generative signature only consume (or reference) nonterminals and LF terms, but their output variables can include LF variables, variables associated with terminals, and variables associated with nonterminals.*

The use of terminal/nonterminal terminology favors the view of generative signatures as context-free grammars, an analogy that holds well for ordered nonterminals. Mobile nonterminals behave more like obligations when we use them as part of the promise-then-fulfill pattern (Section 9.4 and beyond), and persistent nonterminals behave more like constraints.

A generative signature, together with an initial state $(\Psi_0; \Delta_0)$, describes a world with the help of the restriction operator $(\Psi; \Delta)\lightning_\Sigma$ introduced in Section 4.4.2. To recap, if $(\Psi; \Delta)$ is well-defined under the generative signature Σ_{Gen}, and Σ is any signature that includes all of the generative signature's terminals and all of its LF declarations but none of its nonterminals, then $(\Psi; \Delta)\lightning_\Sigma$ is only defined when the only remaining nonterminals in Δ are persistent and can therefore be filtered out of Δ. When the classification of terminals and nonterminals is clear, we will leave off the restricting signature and just write $(\Psi; \Delta)\lightning$.

As a concrete example, let $\mathsf{nt/foo}$ be a persistent nonterminal, let $\mathsf{nt/bar}$ be an ordered nonterminal, and let $\mathsf{t/baz}$ be an ordered terminal. Then $(x{:}\langle \mathsf{nt/bar}\rangle\ ord, y{:}\langle \mathsf{t/baz}\rangle\ ord)\lightning$ is not defined, $(y{:}\langle \mathsf{t/baz}\rangle\ ord)\lightning = (y{:}\langle \mathsf{t/baz}\rangle\ ord)$, and $(x{:}\langle \mathsf{nt/foo}\rangle\ pers, y{:}\langle \mathsf{t/baz}\rangle\ ord)\lightning = (y{:}\langle \mathsf{t/baz}\rangle\ ord)$. Recalling the two-dimensional notation from Chapter 4, we can re-present these three statements as follows:

$$\begin{array}{lll} (x{:}\langle \mathsf{nt/bar}\rangle\ ord, y{:}\langle \mathsf{t/baz}\rangle\ ord) & (y{:}\langle \mathsf{t/baz}\rangle\ ord) & (x{:}\langle \mathsf{nt/foo}\rangle\ pers, y{:}\langle \mathsf{t/baz}\rangle\ ord) \\ \hline\hline & \hline\hline & \hline\hline \\ & (y{:}\langle \mathsf{t/baz}\rangle\ ord) & (y{:}\langle \mathsf{t/baz}\rangle\ ord) \end{array}$$

Definition 9.1 is intentionally quite broad – it need not even be decidable whether a process state belongs to a particular world.[1] Future tractable analyses will therefore presumably be

[1]Proof: consider the initial state $(x{:}\langle \mathsf{gen}\rangle\ ord)$ and the rule $\forall e.\, \forall v.\, \mathsf{gen} \bullet !(\mathsf{ev}\, e\, v) \rightarrowtail \{\mathsf{terminating}\, e\}$. The

based upon further restrictions of the very general Definition 9.1. Context-free grammars are one obvious specialization of generative signatures; we used this correspondence as an intuitive guide in Section 4.4.1. Perhaps less obviously, the regular worlds of Twelf [Sch00] are another specialization of generative signatures.

9.1.1 Regular worlds

The regular worlds used in Twelf [Sch00] are specified with sets of *blocks*. A block describes a little piece of an LF context, and is declared in the LF signature as follows:

$$\mathsf{blockname} : \mathsf{some}\ \{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}\ \mathsf{block}\ \{b_1{:}\tau'_1\} \ldots \{b_m{:}\tau'_m\}$$

A block declaration is well formed in the signature Σ if, by the definition of well-formed signatures from Figure 4.3, $\cdot \vdash_\Sigma a_1{:}\tau_1, \ldots, a_n{:}\tau_n\ \mathsf{ctx}$ and $a_1{:}\tau_1, \ldots, a_{i-1}{:}\tau_n \vdash_\Sigma b_1{:}\tau'_1, \ldots, b_m{:}\tau'_m\ \mathsf{ctx}$.

The first list of LF variable bindings $\{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}$ that come after the some keyword describe the types of concrete LF terms that must exist for the block to be well formed. The second list of LF variable bindings represents the bindings that the block actually adds to the LF context. The regular worlds of Twelf are specified with sets of block identifiers $(\mathsf{block1} \mid \ldots \mid \mathsf{blockn})$. A set $\mathcal{S}$ of block identifiers and a Twelf signature Σ inductively define a world as follows: the empty context belongs to every regular world, and if

- Ψ is a well-formed LF context in the current world,
- $\mathsf{blockname} : \mathsf{some}\ \{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}\ \mathsf{block}\ \{b_1{:}\tau'_1\} \ldots \{b_m{:}\tau'_m\} \in \Sigma$ is one of the blocks in $\mathcal{S}$, and
- there is a σ such that $\Psi \vdash_\Sigma \sigma : a_1{:}\tau_1, \ldots, a_n{:}\tau_n$,

then $\Psi, b_1{:}\sigma\tau'_1, \ldots, b_m{:}\sigma\tau'_m$ is also a well-formed LF context in the current world. The *closed world*, which contains only the empty context, is specified by the empty set of block identifiers.

One simple example of a regular world (previously discussed in Section 4.4.1) is one that contains all contexts with just expression variables of LF type exp. This world can be described with the block blockexp:

$$\mathsf{blockexp} : \mathsf{some}\ \mathsf{block}\ \{x{:}\mathsf{exp}\}$$

If we had a judgment $\mathsf{natvar}\ x\ n$ that associated every LF variable x:exp with some natural number n:nat, then in order to make sure that every expression variable was associated with some natural number we would use the world described by this block:

$$\mathsf{blocknatvar} : \mathsf{some}\ \{n{:}\mathsf{nat}\}\ \mathsf{block}\ \{x{:}\mathsf{exp}\}\ \{nv{:}\mathsf{natvar}\ x\ n\}$$

The world described by the combination of blockexp and blocknatvar is one where every LF variable x:exp is associated with at most one LF variable of type $\mathsf{natvar}\ x\ n$. Assuming that there are no constants of type natvar,[2] this gives us a uniqueness property: if $\mathsf{natvar}\ x\ n$ and $\mathsf{natvar}\ x\ m$, then $m = n$.

---

predicate gen is a nonterminal, the predicate terminating is a terminal, and ev is the encoding of big-step evaluation $e \Downarrow v$ from Figure 6.1. The language described is isomorphic to the set of λ-calculus expressions that terminate under a call-by-value strategy, and membership in that set is undecidable.

[2] This is a property we can easily enforce with subordination, which was introduced in Section 4.1.3.

9.1.2 Regular worlds from generative signatures

A block declaration $\mathsf{blockname} : \mathsf{some}\ \{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}\ \mathsf{block}\ \{b_1{:}\tau'_1\} \ldots \{b_m{:}\tau'_m\}$ can be described by one rule in a generative signature:

$$\mathsf{blockname} : \forall a_1{:}\tau_1 \ldots \forall a_n{:}\tau_n.\ \{\exists b_1{:}\tau'_1 \ldots b_m{:}\tau'_m.\ \mathbf{1}\}$$

Because a regular world is just a set of blocks, the generative signature corresponding to a regular world contains one rule for each block in the regular worlds description. The world described by $(\mathsf{blockexp} \mid \mathsf{blockvar})$ corresponds to the following generative signature:

$\mathsf{nat} : \mathsf{type},$
$\ldots$ *declare constants of type* $\mathsf{nat} \ldots$
$\mathsf{exp} : \mathsf{type},$
$\ldots$ *declare constants of type* $\mathsf{exp} \ldots$
$\mathsf{blockexp} : \{\exists x{:}\mathsf{exp}.\ \mathbf{1}\},$
$\mathsf{blocknatvar} : \forall n{:}\mathsf{nat}.\ \{\exists x{:}\mathsf{exp}.\ \exists nv{:}\mathsf{natvar}\ x\ n.\ \mathbf{1}\}$

Call this regular world signature Σ_{RW}. It is an extremely simple example of a generative signature – there are no terminals and no nonterminals – so the restriction operator has no effect. The world described by $(\mathsf{blockexp} \mid \mathsf{blocknatvar})$ is identical to the set of LF contexts Ψ such that $(\cdot;\cdot) \rightsquigarrow_{\Sigma_{RW}} (\Psi;\cdot)$.

9.1.3 Regular worlds in substructural specifications

It is a simple generalization to replace the proposition $\mathbf{1}$ in the head of the generative block$*$ rules above with less trivial positive SLS propositions. In this way, we can extend the language of regular worlds to allow the introduction of ordered, mobile, and persistent SLS propositions as well. For instance, the rule $\mathsf{blockitem} : \forall n.\ \{\mathsf{item}\ n\}$, where item is a mobile predicate, describes the world of contexts that take the form $(\cdot;\ x_1{:}\langle \mathsf{item}\ n_1\rangle\ eph,\ \ldots,\ x_k{:}\langle \mathsf{item}\ n_k\rangle\ eph)$ for some numbers $n_1 \ldots n_k$. The world described by this generative signature is an invariant of a rule like

$$\mathsf{merge} : \forall n.\ \forall m.\ \forall p.\ \mathsf{item}\ n \bullet \mathsf{item}\ m \bullet\ !(\mathsf{plus}\ n\ m\ p) \rightarrowtail \{\mathsf{item}\ p\}$$

that combines two items, where plus is negative predicate defined with a deductive specification as in Figure 6.21.

Such substructural generalizations of regular worlds are sufficient for the encoding of stores in Linear LF [CP02] and stacks in Ordered LF [Pol01]. They also suffice to describe well-formedness invariants in Felty and Momigliano's sequential specifications [FM12]. However, regular worlds are insufficient for the invariants discussed in the remainder of this chapter.

9.1.4 Generative versus consumptive signatures

Through the example of regular worlds, we can explain why worlds are defined as sets of process states generated by a signature Σ_{Gen} and an initial state $(\Psi;\Delta)$:

$$\{(\Psi';\Delta'') \mid (\Psi;\Delta) \rightsquigarrow^*_{\Sigma_{Gen}} (\Psi';\Delta') \wedge (\Psi';\Delta')\not\rightsquigarrow = (\Psi';\Delta'')\}$$

as opposed to the apparently symmetric case where worlds are sets of process states that *can generate* a final process state $(\Psi; \Delta)$ under a signature $\Sigma Cons$, which we will call a *consumptive* signature:

$$\{(\Psi'; \Delta'') \mid (\Psi'; \Delta') \rightsquigarrow^*_{\Sigma_{Gen}} (\Psi; \Delta) \wedge (\Psi'; \Delta')\!\!\downarrow = (\Psi'; \Delta'')\}$$

Consumptive signatures look like generative signatures with the arrows turned around: we consume well-formed contexts using rules like $\forall e.\, \mathsf{eval}\, e \rightarrowtail \{\mathsf{safe}\}$ and $\forall f.\, \mathsf{safe} \bullet \mathsf{cont}\, f \rightarrowtail \{\mathsf{safe}\}$ instead of creating them with rules like $\forall e.\, \mathsf{gen} \rightarrowtail \{\mathsf{eval}\, e\}$ and $\forall f.\, \mathsf{gen} \rightarrowtail \{\mathsf{gen} \bullet \mathsf{cont}\, f\}$. One tempting property of consumptive signatures is that they open up the possibility of working with complete derivations rather than traces. That is, using a consumptive signature, we can talk about the set of process states $(\Psi; \Delta)$ where $\Psi; \Delta \vdash \mathsf{safe}\ lax$ rather than the set of process states where $(\cdot; x{:}\langle \mathsf{gen} \rangle\ ord) \rightsquigarrow^* (\Psi; \Delta)$.[3]

For purely context-free-grammar-like invariants, such as the PDA invariant from Section 4.4.1 and the SSOS invariant from Section 6.3, generative and consumptive signatures are effectively equivalent. However, for generative signatures describing regular worlds, there is no notion of turning the arrows around to get an appropriate consumptive signature. In particular, say we want to treat

$$\Psi_{good} = (x_1{:}\mathsf{exp}, nv_1{:}\mathsf{natvar}\, x_1\, n_1, x_2{:}\mathsf{exp}, nv_2{:}\mathsf{natvar}\, x_2\, n_2)$$

as a well-formed LF context but *not* treat

$$\Psi_{bad} = (x{:}\mathsf{exp}, nv_1{:}\mathsf{natvar}\, x\, n_1, nv_2{:}\mathsf{natvar}\, x\, n_2)$$

as well-formed. It is trivial to use Twelf's regular worlds or generative signatures to impose this condition, but it does not seem possible to use consumptive signatures for this purpose. There exists a substitution $(x /\!/ x_1, nv_1 /\!/ nv_1, x /\!/ x_2, nv_2 /\!/ nv_2)$ from Ψ_{good} to Ψ_{bad}; therefore, by variable substitution (Theorem 3.4), if there exists a derivation of $\Psi_{good} \vdash_\Sigma \mathsf{gen}\ lax$ there also exists a derivation of $\Psi_{bad} \vdash_\Sigma \mathsf{gen}\ lax$. This is related to the issues of variable and pointer (in)equality discussed in Section 6.5.1.

The generative signatures used to describe state in Section 9.4 and destination-passing style in Section 9.5 rely critically on the uniqueness properties that are provided by generative signatures and not by consumptive signatures.

9.2 Invariants of ordered specifications

We already introduced generative invariants for ordered abstract machine SSOS specifications in Section 6.3. In this section, we will extend that generative invariant to ordered abstract machines with parallel evaluation and recoverable failure.

In Figure 9.1 we define a flat ordered abstract machine with parallel features (parallel evaluation of the function and argument in an application, as discussed in Section 6.1.4 and Figure 6.3) and recoverable failure (as presented in Section 6.5.4 and Figure 6.20). To make sure there

[3] As long as Ψ and Δ contain only nonterminals – using consumptive signatures doesn't obviate the need for the restriction operation $(\Psi; \Delta)\!\!\downarrow$ or some equivalent restriction operation.

```
eval: exp -> prop ord.
retn: exp -> prop ord.
cont: frame -> prop ord.
cont2: frame -> prop ord.
error: prop ord.
handle: exp -> prop ord.

;; Unit
ev/unit: eval unit >-> {retn unit}.

;; Sequential let
ev/let:  eval (let E \x. E' x) >-> {eval E * cont (let1 \x. E' x)}.
ev/let1: retn V * cont (let1 \x. E' x) >-> {eval (E' V)}.

;; Functions and parallel application
ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x)}.
ev/app:  eval (app E1 E2) >-> {eval E1 * eval E2 * cont2 app1}.
ev/app1: retn (lam \x. E x) * retn V2 * cont2 app1
          >-> {eval (E V2)}.

;; Recoverable failure
ev/fail:   eval fail >-> {error}.
ev/catch:  eval (catch E1 E2) >-> {eval E1 * handle E2}.
ev/catcha: retn V * handle _ >-> {retn V}.
ev/catchb: error * handle E2 >-> {eval E2}.

ev/error:  error * cont _ >-> {error}.
ev/errerr: error * error * cont2 _ >-> {error}.
ev/errret: error * retn _ * cont2 _ >-> {error}.
ev/reterr: retn _ * error * cont2 _ >-> {error}.
```

Figure 9.1: Ordered abstract machine with parallel evaluation and failure

is still an interesting sequential feature, we also introduce a let-expression $\ulcorner \mathsf{let}\, x = e\, \mathsf{in}\, e' \urcorner = \mathsf{let} \ulcorner e \urcorner\, \lambda x. \ulcorner e' \urcorner$. The particular features are less important than the general setup, which effectively represents all the specifications from Chapter 6 that used only ordered atomic propositions.

Our goal is to describe a generative signature that represents the well-formed process states of the specification in Figure 9.1. What determines whether a process state is well formed? The intended adequacy theorem was our guide in Section 6.3, and the intended progress theorem will guide our hand in Section 9.3. An obvious minimal requirement is that every state Δ such that $(x{:}\langle \mathsf{eval} \rangle \ulcorner e \urcorner\ ord) \rightsquigarrow^* \Delta$ under the signature from Figure 9.1 must be well formed; otherwise well-formedness won't be invariant under evaluation! One option is therefore to make this correspondence precise, and to have the well formed states be *precisely* the states that are reachable in the process of evaluating syntactically valid expressions $\ulcorner e \urcorner$. That is, if $(x{:}\langle \mathsf{gen} \rangle\ ord) \rightsquigarrow^* \Delta$ under the generative signature and if Δ contains no instances of gen, then there should be an ex-

pression e such that $(x{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle\ ord) \rightsquigarrow^* \Delta$ under the signature from Figure 9.1. (Because gen is the only nonterminal, we can express that Δ contains no instances of gen with the restriction operator, writing $\Delta{\downarrow}$.)[4]

The analogues of the unary grammar productions, associated with the terminals $\mathsf{eval}\ e$, $\mathsf{retn}\ v$, and error, are straightforward:

```
gen/eval:   gen >-> {eval E}.
gen/retn:   gen * !value V >-> {retn V}.
gen/error:  gen >-> {error}.
```

As in Section 6.3, we use a deductively-defined judgment $\mathsf{value}\ v$ to stipulate that we only return values. The process state $(y{:}\langle \mathsf{retn}\ulcorner e_1\ e_2\urcorner\rangle\ ord)$ is not well formed: the application expression $e_1\ e_2$ is not a value, and there is no e such that $(x{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle\ ord) \rightsquigarrow^* (y{:}\langle \mathsf{retn}\ulcorner e_1\ e_2\urcorner\rangle\ ord)$ under the signature from Figure 9.1.

There is a potential catch when we consider the rules for sequential continuations $\mathsf{cont}\ f$ and parallel continuations $\mathsf{cont2}\ f$. We expect a sequential continuation frame to be preceded by a single well-formed computation, and for a parallel continuation frame to be preceded by *two* well-formed computations, suggesting these rules:

```
gen/cont:  gen >-> {gen * cont F}.
gen/cont2: gen >-> {gen * gen * cont2 F}.
```

Even though gen/cont is exactly the rule for sequential continuations in Section 6.3, this approach conflicts with our guiding principle of reachability. Both parallel continuation frames $\mathsf{cont}\ f$ and sequential continuation frames $\mathsf{cont2}\ f$ are indexed by LF terms f of type frame, but the parallel frame app1 cannot appear in a sequential continuation, nor can the sequential frame $(\mathsf{let1}\ \lambda x.e\ x)$ appear in a parallel frame.

This is fundamentally no more complicated than the restrictions we placed on the $\mathsf{retn}\ v$ terminal. All expressions (LF variables of type exp) can appear in $\mathsf{exp}\ e$ propositions (and in $\mathsf{handle}\ e$ propositions), but only some can appear in $\mathsf{retn}\ v$ frames. We describe that subset of frames with the negative atomic proposition $\mathsf{value}\ v$, which is deductively defined. Similarly, only some frames can appear in $\mathsf{cont}\ f$ terminals, and only some frames can appear in $\mathsf{cont2}\ f$ terminals. The former subset can be expressed by a negative atomic proposition $\mathsf{okf}\ f$, and the latter by a negative atomic proposition $\mathsf{okf2}\ f$. Both of these are deductively defined. The full specification of this generative invariant is shown in Figure 9.2; we will refer to this generative signature as $\Sigma_{Gen9.2}$.

9.2.1 Inversion

Traditional inversion lemmas are a critical part of preservation properties for small-step operational semantics specifications. In traditional preservation theorems, we often start with a derivation of $e_1\ e_2 \mapsto e_1'\ e_2$ and another derivation of $\cdot \vdash e_1\ e_2 : tp$. An inversion lemma then proceeds by case analysis on the structure of the derivation $\cdot \vdash e_1\ e_2 : tp$, and allows us to conclude that $\cdot \vdash e_1 : tp' \rightharpoonup tp$ and that $\cdot \vdash e_2 : tp'$ for some object-level type tp'. In other words, an inversion

[4] We won't discuss the proof of this property, but the proof is not difficult to reconstruct; it follows the same contours as the proof of progress given in Chapter 10.

```
value: exp -> prop.
value/unit: value unit.
value/lam: value (lam \x. E x).

okf: frame -> prop.
okf/let1: okf (let1 \x. E' x).

okf2: frame -> prop.
okf2/app1: okf2 app1.

gen: prop ord.
gen/eval:   gen >-> {eval E}.
gen/retn:   gen * !value V >-> {retn V}.
gen/cont:   gen * !okf F >-> {gen * cont F}.
gen/cont2:  gen * !okf2 F >-> {gen * gen * cont2 F}.
gen/error:  gen >-> {error}.
gen/handle: gen >-> {gen * handle E2}.
```

Figure 9.2: Generative invariant: well-formed process states

lemma allows us to take knowledge about a term's structure and obtain information about the structure of typing derivation.

Inversion on a generative signature is intuitively similar: we take information about the structure of a process state and use it to learn about the generative trace that formed that process state. Concurrent equality (Section 4.3) is critical. None of the parts of the lemma below would hold if we did not equate traces such as

$$
\begin{array}{l}
\quad (x'{:}\langle \mathsf{gen} \rangle\ ord) \\
\{x_1, x_2, x_3\} \leftarrow \mathsf{gen/cont2}\ f\ (x' \bullet \mathsf{okf2/app1}) \\
\{y_1\} \leftarrow \mathsf{gen/eval}\ e_1\ x_1 \\
\{y_2\} \leftarrow \mathsf{gen/eval}\ e_2\ x_2 \\
\quad (y_1{:}\langle \mathsf{eval}\ e_1 \rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2 \rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ f \rangle\ ord)
\end{array}
$$

and

$$
\begin{array}{l}
\quad (x'{:}\langle \mathsf{gen} \rangle\ ord) \\
\{x_1, x_2, x_3\} \leftarrow \mathsf{gen/cont2}\ f\ (x' \bullet \mathsf{okf2/app1}) \\
\{y_2\} \leftarrow \mathsf{gen/eval}\ e_2\ x_2 \\
\{y_1\} \leftarrow \mathsf{gen/eval}\ e_1\ x_1 \\
\quad (y_1{:}\langle \mathsf{eval}\ e_1 \rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2 \rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ f \rangle\ ord)
\end{array}
$$

by concurrent equality.

The function of an inversion lemma is to conclude, based on the structure of a generated process state, something about the last step in the trace that generated it. This is less immediate than inversion on derivations because concurrent traces can have many steps which can all

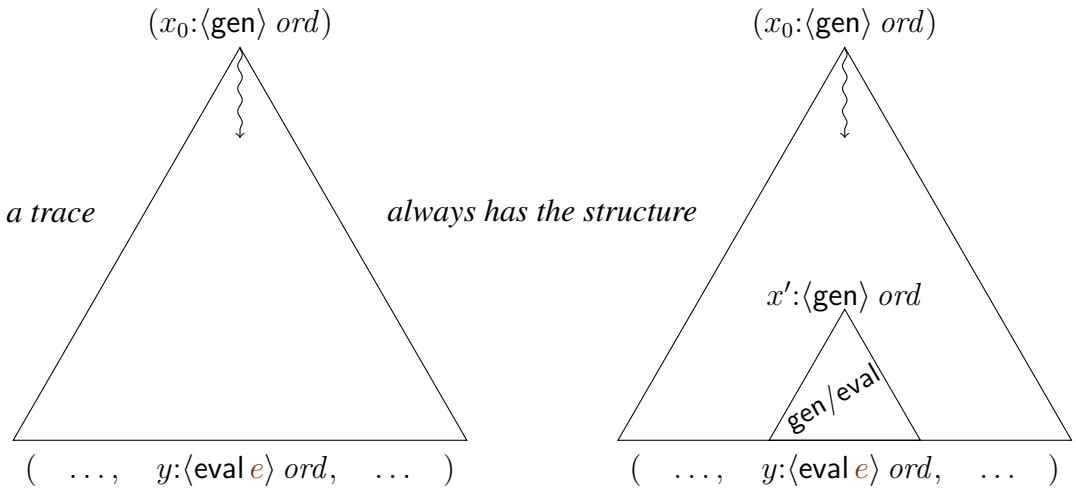


Figure 9.3: Graphical representation of part 1 of the inversion lemma for $\Sigma_{Gen9.2}$

equivalently be treated the last, such as the two gen/eval steps above. Another way of looking at the inversion lemma, which emphasizes that generative traces act like rewriting rules, is shown in Figure 9.3.

Lemma (Inversion – Figure 9.2)**.**

1. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{eval}\ e\rangle\ ord\}$,[5]
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ e\ x')$.
2. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{retn}\ v\rangle\ ord\}$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/retn}\ v\ (x' \bullet\ !N))$,
   where $\cdot \vdash N : \mathsf{value}\ v\ \mathit{true}$.[6]
3. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle\mathsf{gen}\rangle\ ord,\ y_2{:}\langle\mathsf{cont}\ f\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ f\ (x' \bullet\ !N))$,
   where $\cdot \vdash N : \mathsf{okf}\ f\ \mathit{true}$.
4. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle\mathsf{gen}\rangle\ ord,\ y_2{:}\langle\mathsf{gen}\rangle\ ord,\ y_3{:}\langle\mathsf{cont2}\ f\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\ f\ (x' \bullet\ !N))$,
   where $\cdot \vdash N : \mathsf{okf2}\ f\ \mathit{true}$.
5. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{error}\rangle\ ord\}$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/error}\ x')$.
6. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle\mathsf{gen}\rangle\ ord,\ y_2{:}\langle\mathsf{handle}\ e\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/handle}\ e\ (x' \bullet\ !N))$.

In each instance above, $T' :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{x'{:}\langle\mathsf{gen}\rangle\ ord\}$*, where the variables* x_0 *and* x' *may or may not be the same. (They are the same iff* $T' = \diamond$*.)*

[5] Our notation for frames Θ and the tacking-on operation $\Theta\{\Delta\}$ are summarized in Appendix A.

[6] In this chapter, the signature associated with every deductive derivation ($\Sigma_{Gen9.2}$ in this case) is clear from the context and so we write $\cdot \vdash N : \mathsf{value}\ v\ \mathit{true}$ instead of $\cdot \vdash_{\Sigma_{Gen9.2}} N : \mathsf{value}\ v\ \mathit{true}$.

Proof. Each part follows by induction and case analysis on the last steps of T. In each case, we know that the trace cannot be empty, because the variable bindings $y{:}\langle \mathsf{eval}\, e\rangle\ ord$, $y{:}\langle \mathsf{retn}\, v\rangle\ ord$, $y_2{:}\langle \mathsf{cont}\, f\rangle\ ord$, $y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord$, $y{:}\langle \mathsf{error}\rangle\ ord$, and $y_2{:}\langle \mathsf{handle}\, e\rangle\ ord$, respectively, appear in the final process state but not the initial process state. Therefore, $T = T''; S$ for some T'' and S.

There are two ways of formulating the proof of this inversion lemma. The specific formulation does a great deal of explicit case analysis but is closer in style to the preservation lemmas. We also give a more generic formulation of the proof which avoids much of this case analysis, in large part by operating in terms of the input and output interfaces introduced in Section 4.3.

Specific formulation (Part 4)
Given $(T''; S) :: (x_0{:}\langle \mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}$, we perform case analysis on S. We give two representative cases:

Case $S = \{z\} \leftarrow \mathsf{gen/eval}\, e\, x''$
We have that $\Theta\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\} = \Theta'\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\}$. It cannot be the case that $z = y_1$, $z = y_2$, or $z = y_3$ – the propositions don't match. Therefore, we can informally describe the substructural context as a frame Θ_{2H} with two holes that are filled as $\Theta_{2H}\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\}$. (We haven't actually introduced frames with two holes; the reasoning we do with two-hole contexts here could also be done following the structure of the cut admissibility proof, Theorem 3.6.)

If we call the induction hypothesis on T'', we get that

$$
\begin{array}{ll}
T'' = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T''' & \\
& \Theta_{2H}\{x'{:}\langle \mathsf{gen}\rangle\ ord\}\{x''{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\, f\ (x' \bullet\ !N) \\
& \Theta_{2H}\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}\{x''{:}\langle \mathsf{gen}\rangle\ ord\}
\end{array}
$$

The steps $\{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\, f\ (x' \bullet\ !N)$ and $\{z\} \leftarrow \mathsf{gen/eval}\, e\, x''$ can be permuted, so we let $T' = T'''; \{z\} \leftarrow \mathsf{gen/eval}\, e\, x''$ and have

$$
\begin{array}{ll}
T = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T''' & \\
& \Theta_{2H}\{x'{:}\langle \mathsf{gen}\rangle\ ord\}\{x''{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{z\} \leftarrow \mathsf{gen/eval}\, e\, x'' \\
& \Theta_{2H}\{x'{:}\langle \mathsf{gen}\rangle\ ord\}\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\} \\
& \{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\, f\ (x' \bullet\ !N) \\
& \Theta_{2H}\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\}
\end{array}
$$

Case $S = \{z_1, z_2, z_3\} \leftarrow \mathsf{gen/cont2}\, f'\ (x'' \bullet\ !N')$
If $z_1 = x_1$, $z_2 = x_2$, or $z_3 = x_3$, then the ordered structure of the context forces the rest of the equalities to hold and we succeed immediately letting $T' = T''$, $f = f'$, and $N = N'$.

If $z_1 \neq x_1$, $z_2 \neq x_2$, and $z_3 \neq x_3$, then we proceed by induction as in the gen/eval case above.

The only other possibilities allowed by the propositions associated with variables are that $z_1 = x_2$, which is impossible because it would force $z_2 = x_3$ and therefore force gen to equal cont2 f, and that $z_2 = x_1$, which is impossible because it would force $z_3 = x_2$ and therefore force cont2 f' to equal gen.

Generic formulation Let Var be the set of relevant variables – $\{y\}$ in parts 1, 2, and 5, $\{y_1, y_2\}$ in parts 3 and 6, and $\{y_1, y_2, y_3\}$ in part 4.

One possibility is that $\emptyset = S^\bullet \cap Var$. If so, it is always the case that $\emptyset = {}^\bullet S \cap Var$ as well, because Var contains no persistent atomic propositions or LF variables. By the induction hypothesis we then get that $T'' = T'''; S'$, where $S' = \{y\} \leftarrow \mathsf{gen/eval}\, e\, x'$ in part 1, $S' = \{y\} \leftarrow \mathsf{gen/retn}\, v\, (x' \bullet\, !N)$ in part 2, and so on. In each of the six parts, $S'^\bullet = Var$, so $\emptyset = S'^\bullet \cap {}^\bullet S$ and $(T'''; S'; S) = (T'''; S; S')$. We can conclude letting $T' = (T''; S)$.

If, on the other hand, $S^\bullet \cap Var$ is nonempty, we must show by case analysis that $S^\bullet = Var$ and that furthermore S is the step we were looking for. This is easy in parts 1, 2, and 5 where Var is a singleton set: there is only one rule that can produce an atomic proposition of type eval e, retn v, or error, respectively. In part 4, we observe that, if the variable bindings $y_1{:}\langle\mathsf{gen}\rangle\ ord$, $y_2{:}\langle\mathsf{gen}\rangle\ ord$, and $y_3{:}\langle\mathsf{cont2}\ f\rangle\ ord$ appear in order in the substructural context, there is no step in the signature $\Sigma_{Gen9.2}$ that has y_1 among its output variables that does not also have y_2 and y_3 among its output variables, no step that has y_2 among its output variables that does not also have y_1 and y_3 among its output variables, and so on. (This is a rephrasing of the reasoning we did in the gen/cont2 case of the proof above.) Parts 3 and 6 work by similar reasoning. □

The inversion lemma can be intuitively connected with the idea that the grammar described by a generative signature is *unambiguous*. This will not hold in general. If there was a rule $\mathsf{gen/redundant} : \mathsf{gen} \rightarrowtail \{\mathsf{gen}\}$ in $\Sigma_{Gen9.2}$, for instance, then the final step S could be $\{y_1\} \leftarrow \mathsf{gen/redundant}\, y'$, and this would invalidate our inversion lemma for parts 3, 4, and 6.

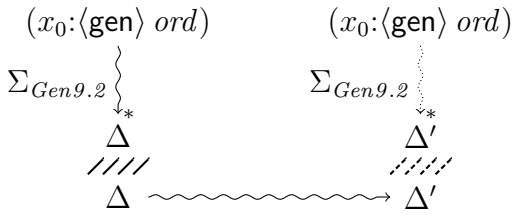
Conversely, if we tried to prove an inversion property about traces $(x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{gen}\rangle\ ord\}$, this property would again fail: $V = \{y\}$, and in the case where the last step S is driven by one of the rules gen/cont, gen/cont2, or gen/handle, $S^\bullet$ will be a strict superset of V.

The reason for preferring the generic formulation to the one based on more straightforward case analysis is that the generic formulation is much more compact. The specific formulation in its full elaboration would require enumerating 7 cases for each of the 6 inversion lemmas, leading to proof whose size is in $O(n^2)$ where n is the number of rules in the generative signature. This enormous proof does very little to capture the intuitive reasons why the steps we are interested in can always be rotated to the end. A goal of this chapter to reason we will emphasize the principles by which we can use to reason *concisely* about specifications.

9.2.2 Preservation

Theorem 9.2 ($\Sigma_{Gen9.2}$ is a generative invariant)**.** *If* $T_1 :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Delta$ *and* $S :: \Delta\!\!\not\downarrow \rightsquigarrow \Delta'$ *under the signature from Figure 9.1, then* $T_2 :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Delta'$.

Again recalling the two-dimensional notation from Chapter 4, the statement of this theorem can be illustrated as follows (dashed lines represent outputs of the theorem):



Proof. By case analysis on S. As in the proofs of Theorem 4.7 and Theorem 6.6, we enumerate the synthetic transitions possible under the signature in Figure 9.1, perform inversion on the structure of T_1, and then use the results of inversion to construct T_2. We give three illustrative cases corresponding to the fragment dealing with functions and parallel application.

Case $\{y\} \leftarrow \mathsf{ev/lam}\ (\lambda x.e)\ x :: \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\} \rightsquigarrow \Theta\{y{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\}$

Applying inversion (Part 1) to T_1, we have

$$
\begin{array}{rl}
T_1 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
\{x\} \leftarrow & \mathsf{gen/eval}\ (\mathsf{lam}\ \lambda x.e)\ x' \\
& \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\}
\end{array}
$$

We can use T' to construct T_2 as follows:

$$
\begin{array}{rl}
T_2 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
\{y\} \leftarrow & \mathsf{gen/retn}\ (\mathsf{lam}\ \lambda x.e)\ (x' \bullet !(\mathsf{value/lam}\ (\lambda x.e))) \\
& \Theta\{y{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\}
\end{array}
$$

Case $\{y_1, y_2, y_3\} \leftarrow \mathsf{ev/app}\ e_1\ e_2\ x$
$:: \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\rangle\ ord\}$
$\rightsquigarrow \Theta\{y_1{:}\langle \mathsf{eval}\ e_1\rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\}$

Applying inversion (Part 1) to T_1, we have

$$
\begin{array}{rl}
T_1 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
\{x\} \leftarrow & \mathsf{gen/eval}\ (\mathsf{app}\ e_1\ e_2)\ x' \\
& \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\rangle\ ord\}
\end{array}
$$

We can use T' to construct T_2 as follows:

$$\begin{array}{ll} T_2 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\ T' & \\ & \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\ & \{y'_1, y'_2, y\} \leftarrow \mathsf{gen/cont2}\ \mathsf{app1}\ (x' \bullet\ !\mathsf{okf2/app1}) \\ & \{y_1\} \leftarrow \mathsf{gen/eval}\ e_1\ y'_1 \\ & \{y_2\} \leftarrow \mathsf{gen/eval}\ e_2\ y'_2 \\ & \Theta\{y_1{:}\langle \mathsf{eval}\ e_1\rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\} \end{array}$$

Case $\{y\} \leftarrow \mathsf{ev/app1}\ (\lambda x.\, e)\ v_2\ (x_1 \bullet x_2 \bullet x_3)$
$:: \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, e)\rangle\ ord,\ x_2{:}\langle \mathsf{retn}\ v_2\rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\}$
$\rightsquigarrow \Theta\{y{:}\langle \mathsf{eval}\ ([v_2/x]e)\rangle\ ord\}$

Applying inversion (Part 2, twice, and then Part 4) to T_1, we have

$$\begin{array}{ll} T_1 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\ T' & \\ & \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\ & \{x'_1, x'_2, x_3\} \leftarrow \mathsf{gen/cont2}\ \mathsf{app1}\ (x' \bullet\ !N) \\ & \{x_1\} \leftarrow \mathsf{gen/retn}\ (\mathsf{lam}\ \lambda x.e)\ (x'_1 \bullet\ !N_1) \\ & \{x_2\} \leftarrow \mathsf{gen/retn}\ v_2\ (x'_2 \bullet\ !N_2) \\ & \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, e)\rangle\ ord,\ x_2{:}\langle \mathsf{retn}\ v_2\rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\} \end{array}$$

We can use T' to construct T_2 as follows:

$$\begin{array}{ll} T_2 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\ T' & \\ & \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\ & \{y\} \leftarrow \mathsf{gen/eval}\ ([v_2/x]e)\ x' \\ & \Theta\{y{:}\langle \mathsf{eval}\ ([v_2/x]e)\rangle\ ord\} \end{array}$$

The other cases, corresponding to the rules ev/unit, ev/fail, ev/catch, ev/catcha, ev/catchb, ev/error, ev/errerr, ev/errret, and ev/reterr all proceed similarly by inversion and reconstruction. □

Note that, in the case corresponding to the rule ev/app1, we obtained but did not use three terms $\cdot \vdash N : \mathsf{okf2}\ \mathsf{app1}\ true$, $\cdot \vdash N_1 : \mathsf{value}\ (\mathsf{lam}\ \lambda x.e)\ true$, and $\cdot \vdash N_2 : \mathsf{value}\ v_2\ true$. By traditional inversion on the structure of a deductive derivation, we know that $N = \mathsf{okf2/app1}$ and $N_1 = \mathsf{value/lam}\ (\lambda x.e)$, but that fact was also not necessary here.

9.3 From well-formed to well-typed states

In order to describe those expressions whose evaluations never get stuck, we introduce object level types tp and define a typing judgment $\Gamma \vdash e : tp$. We encode object-level types as LF terms classified by the LF type typ. The unit type $\ulcorner \mathbf{1} \urcorner = \mathsf{unittp}$ classifies units $\ulcorner \langle\rangle \urcorner = \mathsf{unit}$, and the function type $\ulcorner tp_1 \rightharpoonup tp_2 \urcorner = \mathsf{arr} \ulcorner tp_1 \urcorner \ulcorner tp_2 \urcorner$ classifies lambda expressions.

```
of: exp -> typ -> prop.

of/unit:  of unit unittp.
of/lam:   of (lam \x. E x) (arr Tp' Tp)
           <- (All x. of x Tp' -> of (E x) Tp).
of/app:   of (app E1 E2) Tp
           <- of E1 (arr Tp' Tp)
           <- of E2 Tp'.
of/fail:  of fail Tp.
of/catch: of (catch E1 E2) Tp
           <- of E1 Tp
           <- of E2 Tp.

off: frame -> typ -> typ -> prop.
off/let1: off (let1 \x. E' x) Tp' Tp
           <- (All x. of x Tp' -> of (E' x) Tp).

off2: frame -> typ -> typ -> typ -> prop.
off2/app1:  off2 app1 (arr Tp' Tp) Tp' Tp.

gen: typ -> prop ord.
gen/eval:   gen Tp * !of E Tp >-> {eval E}.
gen/retn:   gen Tp * !of V Tp * !value V >-> {retn V}.
gen/cont:   gen Tp * !off F Tp' Tp >-> {gen Tp' * cont F}.
gen/cont2:  gen Tp * !off2 F Tp1 Tp2 Tp
             >-> {gen Tp1 * gen Tp2 * cont2 F}.
gen/error:  gen Tp >-> {error}.
gen/handle: gen Tp * !of E2 Tp >-> {gen Tp * handle E2}.
```

Figure 9.4: Generative invariant: well-typed process states

In a syntax-directed type system, each syntactic construct is associated with a different typing rule. These are the typing rules necessary for describing the language constructs in Figure 9.1:

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \frac{\Gamma, x{:}tp' \vdash e : tp}{\Gamma \vdash \lambda x.e : tp' \rightharpoonup tp} \qquad \frac{\Gamma \vdash e_1 : tp' \rightharpoonup tp \quad \Gamma \vdash e_2 : tp'}{\Gamma \vdash e_1\, e_2 : tp}$$

$$\frac{}{\Gamma \vdash \mathsf{fail} : tp} \qquad \frac{\Gamma \vdash e_1 : tp \quad \Gamma \vdash e_2 : tp}{\Gamma \vdash \mathsf{try}\, e_1\, \mathsf{ow}\, e_2 : tp}$$

We can adequately encode derivations of the judgment $x_1{:}tp_1, \ldots, x_n{:}tp_n \vdash e : tp$ as SLS derivations $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}; y_1{:}(\mathsf{of}\ x_1\ \ulcorner tp_1 \urcorner)\ pers, \ldots, y_n{:}(\mathsf{of}\ x_1\ \ulcorner tp_n \urcorner)\ pers \vdash \mathsf{of}\ \ulcorner e \urcorner\ \ulcorner tp \urcorner$ under the signature in Figure 9.3.

This typing judgment allows us to describe well-formed initial states, but it is not sufficient to describe intermediate states. To this end, we describe typing rules for frames, refining the negative predicates $\mathsf{okf}\ f$ and $\mathsf{okf2}\ f$ from Figure 9.2. The SLS proposition describing well-typed sequential frames is $(\mathsf{off}\ f\ \ulcorner tp' \urcorner\ \ulcorner tp \urcorner)$. This proposition expresses that the frame f *expects*

a returned result with type tp' and *produces* a computation with type tp.[7] The parallel version is $(\mathsf{off}\ f\ulcorner tp_1\urcorner\ulcorner tp_2\urcorner\ulcorner tp\urcorner)$, and expects two sub-computations with types tp_1 and tp_2, respectively, in order to produce a computation of type tp. These judgments are given in Figure 9.4.

The generative rules in Figure 9.4 are our first use of an *indexed* nonterminal, $\mathsf{gen}\ulcorner tp\urcorner$, which generates computations that, upon successful return, will produce values v such that $\cdot \vdash v : tp$.

9.3.1 Inversion

The structure of inversion lemmas is entirely unchanged, except that it has to account for type indices. We only state two cases of the inversion lemma, the one corresponding to gen/eval and the one corresponding to gen/cont. These two cases suffice to set up the template that all other cases follow.

Lemma (Inversion – Figure 9.2, partial)**.**

1. *If* $T :: (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Theta\{y{:}\langle \mathsf{eval}\ e\rangle\ ord\}$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ tp\ e\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{of}\ e\ tp\ true$

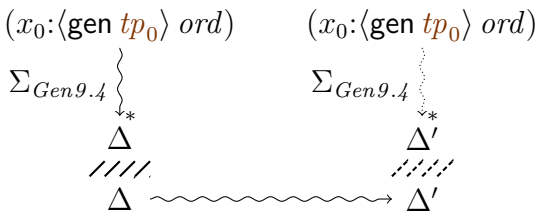
2. *If* $T :: (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Theta\{y_1{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ f\ tp'\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$.

In each instance above, $T' :: (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}$*, where the variables* x_0 *and* x' *may or may not be the same. (They are the same iff* $T' = \diamond$*, and if they are the same that implies* $tp_0 = tp$*.)*

9.3.2 Preservation

Theorem 9.3 only differs from Theorem 9.2 because it mentions the type index. Each object-level type tp_0 describes a different world (that is, a different set of SLS process states), and evaluation under the rules in Figure 9.1 always stays within the same world.

Theorem 9.3 ($\Sigma_{Gen9.4}$ is a generative invariant)**.** *If* $T_1 :: (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Delta$ *and* $S :: \Delta\!\!\downarrow \rightsquigarrow \Delta'$ *under the signature from Figure 9.1, then* $T_2 :: (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Delta'$.



[7] The judgment we encode in SLS as $(\mathsf{off}\ f\ulcorner tp'\urcorner\ulcorner tp\urcorner)$ is written $f : tp' \Rightarrow tp$ in [Har12, Chapter 27].

In the proof of Theorem 9.2, we observed that the applicable inversion on the generative trace gave us derivations like $\cdot \vdash N : \mathsf{okf2}\,\mathsf{app1}\,\mathit{true}$. We did not need these side derivations to complete the proof, but we noted that they were amenable to traditional inversion. Traditional inversion will be critical in proving that the generative invariant described by $\Sigma_{\mathit{Gen9.4}}$ is preserved. It is a solved problem to describe, prove, and mechanize traditional inversion lemmas on deductive derivations; we merely point out when we are using a traditional inversion property in the proof below.

Proof. As always, the proof proceeds by enumeration, inversion, and reconstruction. We give two representative cases:

Case $\{y\} \leftarrow \mathsf{ev/app1}\,(\lambda x.\,e)\,v_2\,(x_1 \bullet x_2 \bullet x_2)$
$:: \Theta\{x_1{:}\langle \mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,e)\rangle\,\mathit{ord},\ x_2{:}\langle \mathsf{retn}\,v_2\rangle\,\mathit{ord},\ x_3{:}\langle \mathsf{cont2}\,\mathsf{app1}\rangle\,\mathit{ord}\}$
$\rightsquigarrow \Theta\{y{:}\langle \mathsf{eval}\,([v_2/x]e)\rangle\,\mathit{ord}\}$

Applying inversion to T_1, we have

$$
\begin{array}{rl}
T_1 = & (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ \mathit{ord}) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ \mathit{ord}\} \\
& \{x'_1, x'_2, x_3\} \leftarrow \mathsf{gen/cont2}\ tp\ \mathsf{app1}\ tp''\ tp'\ (x' \bullet\ !N) \\
& \quad \Theta\{x'_1{:}\langle \mathsf{gen}\ tp''\rangle\ \mathit{ord},\ x'_2{:}\langle \mathsf{gen}\ tp'\rangle\ \mathit{ord},\ x_3{:}\langle \mathsf{cont2}\,\mathsf{app1}\rangle\ \mathit{ord}\} \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\ tp''\ (\mathsf{lam}\ \lambda x.e)\ (x'_1 \bullet\ !N_1 \bullet\ !N_{v1}) \\
& \{x_2\} \leftarrow \mathsf{gen/retn}\ v_2\ (x'_2 \bullet\ !N_2 \bullet\ !N_{v2}) \\
& \quad \Theta\{x_1{:}\langle \mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,e)\rangle\,\mathit{ord},\ x_2{:}\langle \mathsf{retn}\,v_2\rangle\,\mathit{ord},\ x_3{:}\langle \mathsf{cont2}\,\mathsf{app1}\rangle\,\mathit{ord}\}
\end{array}
$$

where

- $\cdot \vdash N : \mathsf{off2}\,\mathsf{app1}\ tp''\ tp'\ tp\ \mathit{true}$.
  By traditional inversion we know $tp'' = \mathsf{arr}\ tp'\ tp$ and $N = \mathsf{off2/app1}\ tp'\ tp$.
- $\cdot \vdash N_1 : \mathsf{of}\ (\mathsf{lam}\ \lambda x.e)\ \mathsf{arr}\ tp'\ tp\ \mathit{true}$.
  By traditional inversion we know $x{:}\mathsf{exp}; dx : \mathsf{of}\ x\ tp'\ \mathit{pers} \vdash N'_1 : \mathsf{of}\ e\ tp\ \mathit{true}$.
- $\cdot \vdash N_2 : \mathsf{of}\ v_2\ tp'$.

With these derivations, variable substitution (Theorem 3.4), and cut admissibility (Theorem 3.6), we have a derivation of $\cdot \vdash [\![N_2/dx]\!]([v_2/x]N'_1) : \mathsf{of}\ ([v_2/x]e)\ tp\ \mathit{true}$.[8] We can therefore use T' to construct T_2 as follows:

$$
\begin{array}{rl}
T_2 = & (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ \mathit{ord}) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ \mathit{ord}\} \\
& \{y\} \leftarrow \mathsf{gen/eval}\ tp\ ([v_2/x]e)\ (x' \bullet\ !([\![N_2/dx]\!]([v_2/x]N'_1))) \\
& \quad \Theta\{y{:}\langle \mathsf{eval}\,([v_2/x]e)\rangle\,\mathit{ord}\}
\end{array}
$$

Case $\{y_1, y_2\} \leftarrow \mathsf{ev/catch}\,(\lambda x.e)\,v_2\,x$
$:: \Theta\{x{:}\langle \mathsf{eval}\,(\mathsf{catch}\,e_1\,e_2)\rangle\,\mathit{ord}\} \rightsquigarrow \Theta\{y_1{:}\langle \mathsf{eval}\,e_1\rangle\,\mathit{ord},\ y_2{:}\langle \mathsf{handle}\,e_2\rangle\,\mathit{ord}\}]$

Applying inversion to T_1, we have

[8] We know by subordination that x is not free in tp, so $[v_2/x]tp = tp$.

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\} \\
& \{x\} \leftarrow \mathsf{gen/eval}\ tp\ (\mathsf{catch}\ e_1\ e_2)\ (x' \bullet\ !N) \\
& \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{catch}\ e_1\ e_2)\rangle\ ord\}
\end{array}
$$

where $\cdot \vdash N : \mathsf{of}\ (\mathsf{catch}\ \mathsf{e}_1\ \mathsf{e}_2)\ tp$.

By traditional inversion on N we know $\cdot \vdash N_1 : \mathsf{of}\ e_1\ tp\ true$ and $\cdot \vdash N_2 : \mathsf{of}\ e_2\ tp\ true$. We can therefore use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\} \\
& \{y'_1, y_2\} \leftarrow \mathsf{gen/handle}\ tp\ e_2\ (x' \bullet\ !N_2) \\
& \{y_1\} \leftarrow \mathsf{gen/eval}\ tp\ e_1\ (y'_1 \bullet\ !N_1) \\
& \Theta\{y_1{:}\langle \mathsf{eval}\ e_1\rangle\ ord,\ y_2{:}\langle \mathsf{handle}\ e_2\rangle\ ord\}
\end{array}
$$

The other cases follow the same pattern. □

Dealing with type preservation is, in an sense, no more difficult than dealing with well-formedness invariants. Theorem 9.3 furthermore follows the contours of a standard progress and preservation proof for an abstract machine like Harper's $\mathcal{K}\{\mathsf{nat}{\rightharpoonup}\}$ [Har12, Chapter 27]. Unlike the on-paper formalism used by Harper, the addition of parallel evaluation in our specification does not further complicate the statement or the proof of the preservation theorem.

9.4 State

Ambient state, encoded in mobile and persistent propositions, was used to describe mutable storage in Section 6.5.1, call-by-need evaluation in Section 6.5.2, and the environment semantics in Section 6.5.3. The technology needed to describe generative invariants for each of these specifications is similar. We will consider the extension of our program from Figure 9.1 with the semantics of mutable storage from Figure 6.14. This specification adds a mobile atomic proposition $\mathsf{cell}\ l\ v$, which the generative signature will treat as a new terminal.

The intuition behind mutable cells is that they exist in tandem with locations l of LF type mutable_loc, giving the non-control part of a process state the following general form:

$$(l_1{:}\mathsf{mutable\_loc}, \ldots, l_n{:}\mathsf{mutable\_loc};\ \langle \mathsf{cell}\ l_1\ v_1\rangle\ eph,\ \ \ldots,\ \langle \mathsf{cell}\ l_n\ v_n\rangle\ eph,\ \ \ldots)$$

Naively, we might attempt to describe such process states with the block-like rule $\mathsf{gen/cell/bad} : \forall v.\ !\mathsf{value}\ v \rightarrowtail \{\exists l.\mathsf{cell}\ l\ v\}$. The problem with such a specification is that it makes cells unable to refer to themselves, a situation that can certainly occur. A canonical example, using back-patching to implement recursion, is traced out in Figure 9.4, which describes a trace classified by:

$(\cdot;\ x_0{:}\langle \mathsf{eval}\ulcorner \mathsf{let}\ f = (\mathsf{ref}\ \lambda x.\langle\rangle)\ \mathsf{in}\ \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e\urcorner\rangle\ ord) \quad \leadsto^*$
$(l_1{:}\mathsf{mutable\_loc};\ y_2{:}\langle \mathsf{cell}\ l_1\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x)\rangle\ eph,\ x_{17}{:}\langle \mathsf{eval}\ [(\mathsf{loc}\ l_1)/f, \mathsf{unit}/x]\ulcorner e\urcorner\rangle\ ord)$

The name of this problem is *parameter dependency* – the term v in gen/cell/bad has to be instantiated before the parameter l is introduced. As a result, the trace in Figure 9.4 includes a step

$$\{x_{16}, y_2\} \leftarrow \mathsf{ev/set2}\ \ldots\ (x_{15} \bullet x_{14} \bullet y_1)$$

that transitions from a state that can be described by Figure 9.2 extended with gen/cell/bad to a state that cannot be described by this signature. This means that gen/cell/bad cannot be the basis of a generative invariant: it's not invariant!

The solution is to create cells in two steps. The first rule, a promise rule, creates the location l and associates a mobile nonterminal gencell with that location. A second fulfill rule consumes that nonterminal and creates the actual mutable cell. Because gencell is a mobile nonterminal, the promise *must* be fulfilled in order for the final state to pass through the restriction operation. As we have already seen, there is not much of a technical difference between well-formedness invariants and well-typedness invariants; Figure 9.6 describes a generative signature that captures type information. This specification introduces two nonterminals. The first is the aforementioned mobile nonterminal gencell l, representing the promise to eventually create a cell corresponding to the location l. The second is a persistent nonterminal ofcell $l\ tp$. The collection of ofcell propositions introduced by a generative trace collectively plays the role of a *store typing* in [Pie02, Chapter 13] or a *signature* in [Har12, Chapter 35]. This promise-then-fulfill pattern appears to be an significant one, and it can be described quite naturally in generative signatures, despite being absent from work on regular-worlds-based reasoning about LF and Linear LF specifications.

9.4.1 Inversion

When we add mutable state, we must significantly generalize the *statement* of inversion lemmas. Derivations and expressions now exist in a world with arbitrary locations l:mutable_loc that are paired with persistent propositions ofcell $l\ tp$.[9]

Lemma (Inversion – Figure 9.6, partial)**.**

1. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi; \Theta\{y{:}\langle \mathsf{eval}\ e\rangle\ ord\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ tp\ e\ (x' \bullet\ !N))$,
   where $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$,
   $T' :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi'; \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}$.
2. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi; \Theta\{y_1{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\})$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ f\ tp'\ (x' \bullet\ !N))$,
   where $\Psi; \Delta \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$,

[9] This purely persistent world fits the pattern of regular worlds. As such, it can be described either with the single rule $\forall tp.\ \{\exists l.\ \mathsf{ofcell}\ l\ tp\}$ or with the equivalent block some tp:typ block l:mutable_loc, $x : \langle \mathsf{ofcell}\ l\ tp\rangle\ pers$.

x_0:⟨eval ⌜let f = (ref $\lambda x.\langle\rangle$) in let x = (f := $\lambda x.(!f)x$) in e⌝⟩

$\{x_1, x_2\} \leftarrow$ ev/let1 ... x_0

x_1:⟨eval ⌜ref $\lambda x.\langle\rangle$⌝⟩, x_2:⟨cont (let1 $\lambda f.$ ⌜let x = (f := $\lambda x.(!f)x$) in e⌝)⟩

$\{x_3, x_4\} \leftarrow$ ev/ref ... x_1

x_3:⟨eval ⌜$\lambda x.\langle\rangle$⌝⟩, x_4:⟨cont ref1⟩, x_2:⟨cont (let1 $\lambda f.$ ⌜let x = (f := $\lambda x.(!f)x$) in e⌝)⟩

$\{x_5\} \leftarrow$ ev/lam ... x_3

x_5:⟨retn ⌜$\lambda x.\langle\rangle$⌝⟩, x_4:⟨cont ref1⟩, x_2:⟨cont (let1 $\lambda f.$ ⌜let x = (f := $\lambda x.(!f)x$) in e⌝)⟩

$\{l_1, x_6, y_1\} \leftarrow$ ev/ref1 ... (x_5 • x_4)

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_6:⟨retn loc l_1⟩, x_2:⟨cont (let1 $\lambda f.$ ⌜let x = (f := $\lambda x.(!f)x$) in e⌝)⟩

$\{x_7\} \leftarrow$ ev/let1 ... (x_6 • x_2)

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_7:⟨eval (let (set (loc l_1) (lam $\lambda x.$ app (get (loc l_1)) x)) $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_8, x_9\} \leftarrow$ ev/let ... x_7

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_8:⟨eval (set (loc l_1) (lam $\lambda x.$ app (get (loc l_1)) x))⟩, x_9:⟨cont (let1 $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_{10}, x_{11}\} \leftarrow$ ev/set ... x_8

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_{10}:⟨eval (loc l_1)⟩, x_{11}:⟨cont (set1 (lam $\lambda x.$ app (get (loc l_1)) x))⟩, x_9:⟨cont (let1 $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_{12}\} \leftarrow$ ev/loc ... x_{10}

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_{12}:⟨retn (loc l_1)⟩, x_{11}:⟨cont (set1 (lam $\lambda x.$ app (get (loc l_1)) x))⟩, x_9:⟨cont (let1 $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_{13}, x_{14}\} \leftarrow$ ev/set1 ... (x_{12} • x_{11})

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_{13}:⟨eval (lam $\lambda x.$ app (get (loc l_1)) x)⟩, x_{14}:⟨cont (set2 l_1)⟩, x_9:⟨cont (let1 $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_{15}\} \leftarrow$ ev/lam ... x_{13}

y_1:⟨cell l_1 ⌜$\lambda x.\langle\rangle$⌝⟩, x_{15}:⟨retn (lam $\lambda x.$ app (get (loc l_1)) x)⟩, x_{14}:⟨cont (set2 l_1)⟩, x_9:⟨cont (let1 $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_{16}, y_2\} \leftarrow$ ev/set2 ... (x_{15} • x_{14} • y_1)

y_2:⟨cell l_1 (lam $\lambda x.$ app (get (loc l_1)) x)⟩, x_{16}:⟨retn unit⟩, x_9⟨cont (let1 $\lambda x.$ ([(loc l_1)/f]⌜e⌝))⟩

$\{x_{17}\} \leftarrow$ ev/let1 ... (x_{16} • x_9)

y_2:⟨cell l_1 (lam $\lambda x.$ app (get (loc l_1)) x)⟩, x_{17}:⟨eval [(loc l_1)/f, unit/x]⌜e⌝⟩

Figure 9.5: Back-patching, with judgments (ord and eph) and arguments corresponding to implicit quantifiers elided

```
ofcell: mutable_loc -> typ -> prop pers.
gencell: mutable_loc -> prop lin.

value/loc: value (loc L).

of/loc: of (loc L) (reftp Tp)
         <- ofcell L Tp.
of/ref: of (ref E) (reftp Tp)
         <- of E Tp.
of/get: of (get E) Tp
         <- of E (reftp Tp).
of/set: of (set E1 E2) unittp
         <- of E1 (reftp Tp)
         <- of E2 Tp.

off/ref1: off ref1 Tp (reftp Tp).
off/get1: off get1 (reftp Tp) Tp.
off/set1: off (set1 E) (reftp Tp) unittp
           <- of E Tp.
off/set2: off (set2 L) Tp unittp
           <- ofcell L Tp.

gencell/promise: {Exists l. !ofcell l Tp * $gencell l}.
gencell/fulfill: $gencell L * !ofcell L Tp * !of V Tp * !value V
                  >-> {$cell L V}.
```

Figure 9.6: Generative invariant: well-typed mutable storage

$T' :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi'; \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\})$,
and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}$.

3. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi; \Theta\{y{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})$,
then $T = (T'; \{y\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp\ v\ (x' \bullet x_t \bullet !N \bullet !N_v))$,
where $x_t{:}\langle \mathsf{ofcell}\ l\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{of}\ v\ tp\ true$, $\Psi; \Delta \vdash N_v : \mathsf{value}\ v\ true$,
$T' :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi'; \Theta\{x'{:}\langle \mathsf{gencell}\ l\rangle\ eph\})$,
and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gencell}\ l\rangle\ eph\}$.

Despite complicating the statement of inversion theorems, the addition of mutable state does nothing to change the structure of these theorems. The new inversion lemma (part 3 above) follows the pattern established in Section 9.2.1.

9.4.2 Uniqueness

To prove that our generative invariant for mutable storage is maintained, we need one property besides inversion; we'll refer to it as the *unique index* property. This is the property that, under the

generative signature described by $\Sigma_{Gen9.6}$, locations always map *uniquely* to persistent positive propositions x_t:ofcell l tp.

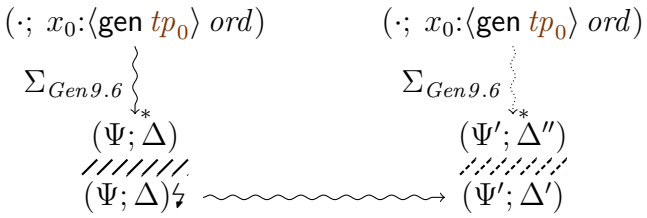
Lemma (Unique indices of $\Sigma_{Gen9.6}$)**.**
1. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi; \Delta)$,
   $x{:}\langle \mathsf{ofcell}\ l\ tp\rangle\ pers \in \Delta$,
   and $y{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers \in \Delta$,
   then $x = y$ *and* $tp = tp'$.

Proof. Induction and case analysis on the last steps of the trace T. □

9.4.3 Preservation

As it was with inversion, the statement of preservation is substantially altered by the addition of locations and mutable state, even though the structure of the proof is not. In particular, because ofcell is a *persistent* nonterminal, we have to expressly represent the fact that the restriction operator $(\Psi; \Delta)\lightning$ will modify the context Δ by erasing the store typing.

Theorem 9.4 ($\Sigma_{Gen9.6}$ is a generative invariant)**.** *If* $T_1 :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi; \Delta)$ *and* $S :: (\Psi; \Delta)\lightning \leadsto (\Psi'; \Delta')$ *under the combined signature from Figure 9.1 and Figure 6.14, then* $(\Psi'; \Delta') = (\Psi'; \Delta'')\lightning$ *for some* Δ'' *such that* $T_2 :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.6}} (\Psi'; \Delta'')$.



Proof. As always, the proof proceeds by enumeration, inversion, and reconstruction. The only interesting cases are the three that actually manipulate state, corresponding to ev/ref1, ev/get1, and ev/set2. Recall these three rules from Figure 6.14:

```
ev/ref1: retn V * cont ref1
          >-> {Exists l. $cell l V * retn (loc l)}.
ev/get1: retn (loc L) * cont get1 * $cell L V
          >-> {retn V * $cell L V}.
ev/set2: retn V2 * cont (set2 L) * $cell L _
          >-> {retn unit * $cell L V2}.
```

Reasoning about the last two cases is similar, so we only give the cases for ev/ref and ev/get1 below.

Case $\{l, y_1, y_2\} \leftarrow \mathsf{ev/ref1}\ v\ (x_1 \bullet x_2)$
$:: (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\})$
$\leadsto (\Psi,\ l{:}\mathsf{mutable\_loc};\ \Theta\{y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ y_2{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord\})$

$T_1 :: (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \leadsto^* (\Psi;\ \Theta'\{x_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\})$ for some Θ' such that, for all Ξ, $(\Psi; \Theta'\{\Xi\})\lightning = (\Psi; \Theta\{\Xi\lightning\})$. Applying inversion to T_1, we have

$$
\begin{array}{ll}
T_1 = & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}) \\
& \{x'_1, x_2\} \leftarrow \mathsf{gen/cont}\ tp\ \mathsf{ref1}\ tp'\ (x' \bullet\ !N) \\
& (\Psi;\ \Theta'\{x'_1{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\}) \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\ v\ (x'_1 \bullet\ !N_1 \bullet\ !N_{v1}) \\
& (\Psi;\ \Theta'\{x_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\})
\end{array}
$$

where Δ contains the persistent propositions from Θ' and where

- $\Psi; \Delta \vdash N : \mathsf{off}\ \mathsf{ref1}\ tp'\ tp\ true$. By traditional inversion we know $tp = \mathsf{reftp}\ tp'$ and $N = \mathsf{off/ref1}\ tp'$.
- $\Psi; \Delta \vdash N_1 : \mathsf{of}\ v\ tp'\ true$.
- $\Psi; \Delta \vdash N_{v1} : \mathsf{value}\ v\ true$.

We can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_2 = & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ (\mathsf{reftp}\ tp')\rangle\ ord\}) \\
& \{l, z, y'_1\} \leftarrow \mathsf{gencell/promise}\ tp' \\
& \{y_1\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp'\ v\ (y'_1 \bullet z \bullet\ !N_1 \bullet\ !N_{v1}) \\
& (\Psi, l{:}\mathsf{mutable\_loc}; \\
& \quad \Theta'\{z{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers,\ y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ x'{:}\langle \mathsf{gen}\ (\mathsf{ref}\ tp')\rangle\ ord\}) \\
& \{y_2\} \leftarrow \mathsf{gen/retn}\ (\mathsf{reftp}\ tp')\ (\mathsf{loc}\ l)\ (x' \bullet\ !(\mathsf{of/loc}\ l\ tp'\ z) \bullet\ !(\mathsf{value/loc}\ l)) \\
& (\Psi, l{:}\mathsf{mutable\_loc}; \\
& \quad \Theta'\{z{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers,\ y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ x'{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord\})
\end{array}
$$

Restriction removes the persistent nonterminal $z{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers$ from the context, so the restriction of T_2's output is $(\Psi,\ l{:}\mathsf{mutable\_loc};\ \Theta\{y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ y_2{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord\})$ as required.

Case $\{y_1, y_2\} \leftarrow \mathsf{ev/get1}\ l\ v\ (x_1 \bullet x_2 \bullet x_3)$
$:: (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle\ ord,\ x_3{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})$
$\leadsto (\Psi;\ \Theta\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})$

$T_1 :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord)$
$\leadsto^* (\Psi;\ \Theta'\{x_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle\ ord,\ x_3{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})$
for some Θ' such that, for all Ξ, $(\Psi; \Theta'\{\Xi\})\lightning = (\Psi; \Theta\{\Xi\lightning\})$. Applying inversion to T_1, we have

$$
\begin{array}{ll}
T_1 = & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ x'_3{:}\langle \mathsf{gencell}\ l\rangle\ eph\}) \\
& \{x_3\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp'\ v\ (x'_3 \bullet z_1 \bullet\ !N_3 \bullet\ !N_{v3})
\end{array}
$$

```
gencount/finalize: $gencount N >-> {$counter N}.

gencell/promise: $gencount N
                  >-> {Exists l.
                       !ofcell l Tp * $gencell l N * $gencount (s N)}.

gencell/fulfill: $gencell L N * !ofcell L Tp * !of V Tp * !value V
                  >-> {$cell L N V}.
```

Figure 9.7: Generative invariants for cells with unique natural-number tags

$$
\begin{array}{l}
\qquad (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ x_3{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\}) \\
\{x'_1, x_2\} \leftarrow \mathsf{gen/cont}\ tp\ \mathsf{get1}\ tp''\ (x' \bullet\ !N_2) \\
\{x_1\} \leftarrow \mathsf{gen/retn}\ tp''\ (\mathsf{loc}\ l)\ (x'_1 \bullet\ !N_1 \bullet\ !N_{v1}) \\
\qquad (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle\ ord,\ x_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})
\end{array}
$$

where Δ contains the persistent propositions from Θ' and where

- $\Psi; \Delta \vdash N_2 : \mathsf{off}\ \mathsf{get1}\ tp''\ tp\ true$. By traditional inversion we know $tp'' = \mathsf{reftp}\ tp$ and $N_2 = \mathsf{off/get1}\ tp$.
- $\Psi; \Delta \vdash N_1 : \mathsf{of}\ (\mathsf{loc}\ l)\ (\mathsf{reftp}\ tp)\ true$. By traditional inversion we know $N_1 = \mathsf{of/loc}\ l\ tp\ x''_1$ where $x''_1{:}\langle \mathsf{ofcell}\ l\ tp\rangle\ pers \in \Delta$.
- $x'_3{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers \in \Delta$.
- $\Psi; \Delta \vdash N_3 : \mathsf{of}\ v\ tp'\ true$.
- $\Psi; \Delta \vdash N_{3v} : \mathsf{value}\ v\ true$.

By the uniqueness lemma, we have that $x'_3 = x''_1$ and $tp' = tp$. Therefore, we can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_1 = & \qquad (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
& T' \\
& \qquad (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ x'_3{:}\langle \mathsf{gencell}\ l\rangle\ eph\}) \\
& \{y_2\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp\ v\ (x'_3 \bullet\ z_1 \bullet\ !N_3 \bullet\ !N_{v3}) \\
& \qquad (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\}) \\
& \{y_1\} \leftarrow \mathsf{gen/retn}\ tp\ v\ (x' \bullet\ !N_3 \bullet\ !N_{v3}) \\
& \qquad (\Psi;\ \Theta'\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})
\end{array}
$$

$(\Psi;\ \Theta\{y_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l), y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\rangle\ ord\})$ is the restriction of T_2's output, as required.

The other cases, notably ev/set2, follow the same pattern. □

9.4.4 Revisiting pointer inequality

As we discussed in Section 6.5.1, the fact that SLS variables cannot be directly checked for inequality complicates the representation of languages that can check for the inequality of locations. One way of circumventing this shortcoming is by keeping a runtime counter in the form of an ephemeral atomic proposition count n that counts the number of currently allocated cells; the

```
gen: typ -> dest -> prop lin.

gen/dest: {Exists d:dest. one}.
gen/eval: $gen T D * !of E T >-> {$eval E D}.
gen/retn: $gen T D * !of V T * !value V >-> {$retn V D}.
gen/cont: $gen T D * !off F T' T
           >-> {Exists d'. gen T' d' * $cont F d' D}.
```

Figure 9.8: Generative invariant: destination-passing ("obvious" formulation)

rule ev/ref1 that allocates a new cell must be modified to access and increment this counter and to attach the counter's value to the new cell. Inequality of those natural number tags can then be used as a proxy for inequality of locations.

A generative signature like the one in Figure 9.7 could be used to represent the invariant that each location and each cell is associated with a unique natural number. The techniques described in this chapter should therefore be sufficient to describe generative invariants of SSOS specifications that implement pointer inequality in this way.

9.5 Destination-passing

Destination-passing style specifications, as discussed in Chapter 7, are not a focus of this dissertation, but they deserve mention for two reasons. First, they are of paramount importance in the context of the logical framework CLF, a framework that lacks SLS's notions of order. Second, the work of Cervesato and Sans [CS13] is the most closely related work on describing progress and preservation properties for SSOS-like specifications; their work closely resembles a destination-passing specification. As such, the preservation property given in this section can be viewed an encoding of the proof by Cervesato and Sans in SLS.

In this section, we will work with an operational semantics derived from the signature given in Figure 7.5 (sequential evaluation of function application) in Chapter 7. To use sequential application instead of parallel evaluation of function application, we will need to give different typing rules for frames:

```
off/app1: off app1 Tp (app1 E) (arr Tp' Tp) Tp
           <- of E Tp'.
off/app2: off (app2 \x. E x) Tp' Tp
           <- (All x. of x Tp' -> of (E x) Tp).
```

Other than this change, our deductive typing rules stay the same.

When we move from ordered abstract machines to destination-passing style, the most natural adaptation of generative invariants is arguably the one given in Figure 9.8. In that figure, the core nonterminal is the mobile proposition gen $tp\ d$. The rule gen/dest, which creates destinations

```
gen: typ -> dest -> prop lin.
gendest: dest -> dest -> prop lin.

dest/promise: {Exists d'. $gendest d' D}.
dest/unused:  ($gendest D' D) >-> {one}.

gen/eval: $gen Tp D * !of E Tp >-> {$eval E D}.
gen/retn: $gen Tp D * !of V Tp * !value V >-> {$retn V D}.
gen/cont: $gen Tp D * !off F Tp' Tp * $gendest D' D
           >-> {$gen Tp' D' * $cont F D' D}.
```

Figure 9.9: Generative invariant: destination-passing (modified formulation)

freely, is necessary, as we can see from the following sequence of process states:

$$
\begin{aligned}
(d_0{:}\mathsf{dest};\ & x_1{:}\langle \mathsf{eval}\ulcorner(\lambda x.e)\ e_2\urcorner\ d_0\rangle\ eph) \rightsquigarrow \\
(d_0{:}\mathsf{dest}, d_1{:}\mathsf{dest};\ & x_2{:}\langle \mathsf{eval}\ulcorner(\lambda x.e)\urcorner\ d_1\rangle\ eph,\ x_3{:}\langle \mathsf{cont}\ (\mathsf{app1}\ulcorner e_2\urcorner)\ d_1\ d_0\rangle\ eph) \rightsquigarrow \\
(d_0{:}\mathsf{dest}, d_1{:}\mathsf{dest};\ & x_4{:}\langle \mathsf{retn}\ulcorner(\lambda x.e)\urcorner\ d_1\rangle\ eph,\ x_3{:}\langle \mathsf{cont}\ (\mathsf{app1}\ulcorner e_2\urcorner)\ d_1\ d_0\rangle\ eph) \rightsquigarrow \\
(d_0{:}\mathsf{dest}, d_1{:}\mathsf{dest}, d_2{:}\mathsf{dest};\ & x_5{:}\langle \mathsf{eval}\ulcorner e_2\urcorner\ d_2\rangle\ eph,\ x_6{:}\langle \mathsf{cont}\ (\mathsf{app2}\ulcorner\lambda x.e\urcorner)\ d_2\ d_0\rangle\ eph) \rightsquigarrow \ldots
\end{aligned}
$$

In the final state, d_1 is isolated, no longer mentioned anywhere else in the process state, so gen/dest must be used in the generative trace showing that the last state above is well-typed.

We will not use the form described in Figure 9.8 in this chapter, however. Instead, we will prefer the presentation in Figure 9.9. There are two reasons for this. First and foremost, this formulation meshes better with the promise-then-fulfill pattern that was necessary for state in Figure 9.6 and that is also necessary for continuations in Section 9.6 below. As a secondary consideration, using the first formulation would require us to significantly change the structure of our inversion lemmas. In previous inversion lemmas, proving that gen/cont could always be rotated to the end of a generative trace was simple, because it introduced no LF variables or persistent nonterminals. The gen/cont rule in Figure 9.8 does introduce an LF variable d', invalidating the principle used in Section 9.2.1.

The dest/promise rule in Figure 9.9 is interesting in that it requires each destination d' to be created along with foreknowledge of the destination, d, that the destination d' will return to. This effectively forces all the destinations into a tree structure from the moment of their creation onwards, a point that will become important when we modify Figure 9.9 to account for persistent destinations and first-class continuations. The root of this tree is the destination d_0 that already exists in the initial process state $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph)$.

9.5.1 Uniqueness and index sets

One consequence of the way we use the promise-then-fulfill pattern in this specification is that our unique index property becomes conceptually prior to our inversion lemma.

Lemma (Unique indices of $\Sigma_{Gen9.9}$)**.**

1. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gendest}\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{gendest}\ d\ d_2\rangle\ eph \in \Delta$, *then* $x = y$ *and* $d_1 = d_2$.
2. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gendest}\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph \in \Delta$, *then there is a contradiction.*
3. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gendest}\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{cont}\ f\ d\ d'\rangle\ eph \in \Delta$, *then there is a contradiction.*
4. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gen}\ tp_1\ d\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{gen}\ tp_2\ d\rangle\ eph \in \Delta$, *then* $x = y$ *and* $tp_1 = tp_2$.
5. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{cont}\ f_1\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{cont}\ f_2\ d\ d_2\rangle\ eph \in \Delta$, *then* $x = y$, $f_1 = f_2$, *and* $d_1 = d_2$.

Proof. Induction and case analysis on last steps of the trace T; each part uses the previous parts (parts 2 and 3 use part 1, and parts 4 and 5 use parts 2 and 3). □

This lemma is a lengthy way of expressing what is ultimately a very simple property: that the second position of gendest is a unique index and that it passes on that unique indexing to the second position of gen and the second position of cont.

Definition 9.5. *A set* S *is a* unique index set *under a generative signature* Σ *and an initial state* $(\Psi; \Delta)$ *if, whenever*

- $\mathsf{a}/i \in S$,
- $\mathsf{b}/j \in S$,
- $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi'; \Delta')$,
- $x{:}\langle \mathsf{a}\ t_1 \ldots t_n\rangle\ lvl \in \Delta'$, *and*
- $y{:}\langle \mathsf{b}\ s_1 \ldots s_m\rangle\ lvl' \in \Delta'$,

it is the case that $t_i = s_j$ *implies* $x = y$. *Of course, if* $x = y$, *that in turn implies that* $\mathsf{a} = \mathsf{b}$, $i = j$, $n = m$, $t_k = s_k$ *for* $1 \leq k \leq n$, *and* $lvl = lvl'$.

The complicated lemma above can then be captured by the dramatically more concise statement: $\{\mathsf{gendest}/1, \mathsf{gen}/2\}$ and $\{\mathsf{gendest}/1, \mathsf{cont}/2\}$ are both unique index sets under the signature $\Sigma_{Gen9.9}$ and the initial state $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph)$. In fact, we can extend the first unique index set to $\{\mathsf{gendest}/1, \mathsf{gen}/2, \mathsf{eval}/2, \mathsf{retn}/2\}$. Stating that $\{\mathsf{gendest}/1, \mathsf{gen}/2\}$ was a unique index property previously required 3 distinct statements, and it would take 10 distinct statements to express that $\{\mathsf{gendest}/1, \mathsf{gen}/2, \mathsf{eval}/2, \mathsf{retn}/2\}$ is a unique index property.[10] The

[10] Four positive statements (similar to parts 1, 4, and 5 of the lemma above) along $\binom{4}{2} = 6$ negative ones (similar to parts 2 and 3 of the lemma above).

unique index property for cells (Section 9.4.2) can be rephrased by saying that $\{\mathsf{ofcell}/1\}$ is a unique index set; $\{\mathsf{gencell}/1, \mathsf{cell}/1\}$ is also a unique index set in that specification.

It's also possible for unique index sets to be simply (and, presumably, mechanically) checked. This amounts to a very simple preservation property.

9.5.2 Inversion

Lemma (Inversion – Figure 9.9)**.**

1. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{y{:}\langle \mathsf{eval}\ e\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ tp\ d\ e\ (x' \bullet\ !N))$,
   where $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\}$.

2. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{y{:}\langle \mathsf{retn}\ v\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/retn}\ tp\ d\ v\ (x' \bullet\ !N \bullet\ !N_v))$,
   where $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$, $\Psi; \Delta \vdash N_v : \mathsf{value}\ v\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\}$.

3. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{y_1{:}\langle \mathsf{gen}\ tp'\ d'\rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ f\ d'\ d\rangle\ eph\})$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ f\ tp'\ d'\ (x' \bullet\ !N \bullet\ z))$,
   where $\Psi; \Delta \vdash N : \mathsf{off}\ e\ tp'\ tp\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\}$.

Proof. As with other inversion lemmas, each case follows by induction and case analysis on the last steps of T. The trace cannot be empty, so $T = T''; S$ for some T'' and S, and we let Var be the set of relevant variables $\{y\}$ in parts 1 and 2, and $\{y_1, y_2\}$ in part 3.

If $\emptyset = S^\bullet \cap Var$, the proof proceeds by induction as it did in Section 9.2.1.

If $S^\bullet \cap Var$ is nonempty, then we must again show by case analysis that $S^\bullet = Var$ and that furthermore S is the step we were looking for. As before, this is easy for the unary grammar productions where Var is a singleton set: there is only one rule that can produce an atomic proposition $\mathsf{eval}\ e\ d$ or $\mathsf{retn}\ v\ d$.

When Var is not a singleton (which only happens in part 3 for this lemma), we must use the unique index property to reason that if there is any overlap, that overlap must be total.

- Say $S = \{y_1, y_2''\} \leftarrow \mathsf{gen/cont}\ tp\ d''\ f''\ tp'\ d'\ (x' \bullet\ !N \bullet\ z)$.
  Then the final state contains $y_2{:}\langle \mathsf{cont}\ f\ d'\ d\rangle\ eph$ and $y_2''{:}\langle \mathsf{cont}\ f''\ d'\ d''\rangle\ eph$. The shared d' and the unique index property ensures that $y_2 = y_2''$, $f = f''$, and $d = d''$.

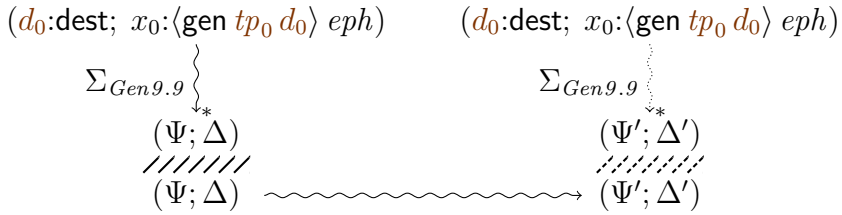
$*$ Say $S = \{y_1'', y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ f\ tp'''\ d'\ (x' \bullet\ !N \bullet z)$.
Then the final state contains $y_1{:}\langle \mathsf{gen}\ tp'\ d' \rangle\ eph$ and $y_1''{:}\langle \mathsf{gen}\ tp'''\ d' \rangle\ eph$. The shared d' and the unique index property ensures that $y_1 = y_1''$ and $tp' = tp'''$.

Therefore, $S = \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ f\ tp'\ d'\ (x' \bullet\ !N \bullet z)$. □

9.5.3 Preservation

As we once again have no persistent nonterminals, we can return to the simpler form of the preservation theorem used in Theorem 9.2 and Theorem 9.3 (compared to the more complex formulation needed for Theorem 9.4).

Theorem 9.6 ($\Sigma_{Gen9.9}$ is a generative invariant)**.** *If* $T_1 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$ *and there is a step* $S :: (\Psi; \Delta) \not\rightsquigarrow (\Psi'; \Delta')$ *under the signature from Figure 7.5, then* $T_2 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.9}} (\Psi'; \Delta')$.



Proof. As usual, we enumerate the synthetic transitions possible under the signature in Figure 7.5, perform inversion on the structure of T_1, and then use the results of inversion to construct T_2. We give one illustrative case.

Case $\{d_2, y_1, y_2\} \leftarrow \mathsf{ev/app1}\ (\lambda x.e)\ d_1\ e_2\ d\ (x_1 \bullet x_2)$
$:: (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\ d_1 \rangle\ eph,\ x_2{:}\langle \mathsf{cont}\ (\mathsf{app1}\ e_2)\ d_1\ d \rangle\ eph\})$
$\rightsquigarrow (\Psi, d_2{:}\mathsf{dest};\ \Theta\{y_1{:}\langle \mathsf{eval}\ e_2\ d_2 \rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ (\mathsf{app2}\ \lambda x.e)\ d_2\ d \rangle\ eph\})$

Applying inversion (Part 2, then Part 3) to T_1, we have

$$
\begin{array}{ll}
T_1 = & (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \\
T' & \\
& (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d \rangle\ eph,\ z_1{:}\langle \mathsf{gendest}\ d_1\ d \rangle\ eph\}) \\
& \{x_1', x_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ (\mathsf{app1}\ e_2)\ tp'\ d_1\ (x' \bullet\ !N_2 \bullet z_1) \\
& (\Psi;\ \Theta\{x_1'{:}\langle \mathsf{gen}\ tp'\ d_1 \rangle\ eph,\ x_2{:}\langle \mathsf{cont}\ (\mathsf{app1}\ e_2)\ d_1\ d \rangle\ eph\}) \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\ tp'\ d_1\ v\ (x_1' \bullet\ !N_1 \bullet\ !N_{v1}) \\
& (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\ d_1 \rangle\ eph,\ x_2{:}\langle \mathsf{cont}\ (\mathsf{app1}\ e_2)\ d_1\ d \rangle\ eph\})
\end{array}
$$

where Δ contains the persistent propositions from Θ and where

- $\Psi; \Delta \vdash N_2 : \mathsf{off}\ (\mathsf{app1}\ e_2)\ tp'\ tp\ true$. By traditional inversion we know there exists tp'' and N_2' such that $tp' = \mathsf{arr}\ tp''\ tp$ and $\Psi; \Delta \vdash N_2' : \mathsf{of}\ e_2\ tp''\ true$.
- $\Psi; \Delta \vdash N_1 : \mathsf{of}\ (\mathsf{lam}\ \lambda x.e)\ \mathsf{arr}\ tp'\ tp\ true$. By traditional inversion we know there exists N_1' where $\Psi, x{:}\mathsf{exp}; \Delta, dx : (\mathsf{of}\ x\ tp')\ pers \vdash N_1' : \mathsf{of}\ e\ tp\ true$.

We can use T' to construct T_2 as follows:

```
offuture: exp -> typ -> prop pers.
genfuture: dest -> exp -> prop lin.

of/future: of X Tp <- offuture X Tp.

future/promise: {Exists d. Exists x. $genfuture d x * !offuture x Tp}.

future/compute: $genfuture D X * !offuture X Tp
                 >-> {$gen Tp D * $promise D X}.

future/bind:    $genfuture D X * !offuture X Tp * !of V Tp * !value V
                 >-> {!bind X V}.
```

Figure 9.10: Generative invariant: futures

$$
\begin{array}{lll}
T_2 = & & (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \\
& T' & \\
& & (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z_1{:}\langle \mathsf{gendest}\ d_1\ d\rangle\ eph\}) \\
& & \{\} \leftarrow \mathsf{dest/unused}\ d_1\ d\ z_1 \\
& & \{d_2, z_2\} \leftarrow \mathsf{dest/promise}\ d \\
& & (\Psi, d_2{:}\mathsf{dest};\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z_2{:}\langle \mathsf{gendest}\ d_2\ d\rangle\ eph\}) \\
& & \{y_1', y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ (\mathsf{app2}\lambda x.e)\ tp''\ d_2 \\
& & \qquad\qquad (x' \bullet !(\mathsf{off/app2}\ tp''\ (\lambda x.e)\ tp\ (\lambda x, dx.\, !N_1')) \bullet z_2) \\
& & (\Psi, d_2{:}\mathsf{dest};\ \Theta\{y_1'{:}\langle \mathsf{gen}\ tp''\ d_2\rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ (\mathsf{app2}\ \lambda x.e)\ d_2\ d\rangle\ eph\}) \\
& & \{y_1, y_2\} \leftarrow \mathsf{gen/eval}\ tp''\ d_2\ e_2\ (y_1' \bullet !N_2') \\
& & (\Psi, d_2{:}\mathsf{dest};\ \Theta\{y_1{:}\langle \mathsf{eval}\ e_2\ d_2\rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ (\mathsf{app2}\ \lambda x.e)\ d_2\ d\rangle\ eph\})
\end{array}
$$

The other cases follow the same pattern. □

9.5.4 Extensions

Generative invariants for parallel evaluation (Figure 7.6) and the alternate semantics of parallelism and failure (Figure 7.7) as described in Section 7.2.1 are straightforward extensions of the development in this section. Synchronization (Section 7.2.2) and futures (Section 7.2.3) are a bit more interesting from the perspective of generative invariants and preservation. Figure 9.10 is one proposal for a generative invariant for our SLS encoding of futures, but we leave further consideration for future work.

9.6 Persistent continuations

The final specification style we will cover in detail is the use of persistent continuations as discussed in Section 7.2.4 as a way of giving an SSOS semantics for first-class continuations (Figure 7.11). The two critical rules from Figure 7.11 are repeated below: ev/letcc captures the

```
gen: prop lin.
ofdest: dest -> typ -> prop pers.
gendest: dest -> dest -> prop lin.

value/contn: value (contn D).

of/letcc: of (letcc \x. E x) Tp
           <- (All x. of x (conttp Tp) -> of (E x) Tp).
of/contn: of (contn D) (conttp Tp)
           <- ofdest D Tp.
of/throw: of (throw E1 E2) Tp'
           <- of E1 Tp
           <- of E2 (conttp Tp).

off/throw1: off (throw1 E2) Tp Tpx
             <- of E2 (conttp Tp).
off/throw2: off (throw2 V1) (conttp Tp) Tpx
             <- of V1 Tp
             <- value V1.

dest/promise: {Exists d'. $gendest d' D * !ofdest d' Tp'}.
dest/fulfill: $gendest D' D *
              !off F Tp' Tp * !ofdest D' Tp' * !ofdest D Tp
               >-> {!cont F D' D}.

gen/eval: $gen * !ofdest D Tp * !of E Tp >-> {eval E D}.
gen/retn: $gen * !ofdest D Tp * !of V Tp * !value V >-> {retn V D}.
```

Figure 9.11: Generative invariant: persistent destinations and first-class continuations

destination representing the current continuation and the rule ev/throw2 throws away the continuation represented by d_2 and throws computation to the continuation represented by the dk.

```
ev/letcc:  $eval (letcc \x. E x) D >-> {$eval (E (contn D)) D}.
ev/throw2: $retn (contn DK) D2 * !cont (throw2 V1) D2 D
            >-> {$retn V1 DK}.
```

While the setup of Figure 9.9 is designed to make the transition to persistent continuations and letcc seem less unusual, this section still represents a radical shift.

It should not be terribly surprising that the generative invariant for persistent continuations is rather different than the other generative invariants. Generative invariants capture patterns of specification, and we have mostly concentrated on patterns that facilitate concurrency and communication. Persistent continuations, on the other hand, are a pattern mostly associated with first-class continuations. There is not an obvious way to integrate continuations and parallel or concurrent evaluation, and the proposal by Moreau and Ribbens in [MR96] is not straightforward to adapt to the semantic specifications we gave in Chapter 7.

Consider again the gendest/promise rule from Figure 9.9. The rule consumes no nonterminals and is the only rule introducing LF variables, so any $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^* (\Psi; \Delta)$ under $\Sigma_{Gen9.9}$ can be factored into two parts $T = T_1; T_2$ where T_1 contains only steps that use gendest/promise. The computational effect of Theorem 9.6 is that T_1 grows to track the tree-structured shape of the stack, both past and present. We could record, if we wanted to, the *past* structure of the control stack by adding a persistent nonterminal ghostcont $f\ d'\ d$ and modifying dest/unused in Figure 9.9 as follows:

```
dest/unused:  $gendest D' D >-> {!ghostcont F D' D}.
```

Once we make the move to persistent continuations, however, there's no need to create a ghost continuation, we can just have the rule dest/unused (renamed to dest/fulfill in Figure 9.11) create the continuation itself. To make this work, dest/promise predicts the type that will be associated with a newly-generated destination d by generating a persistent nonterminal ofdest $d\ tp$. (This is just like how gencell/promise in Figure 9.6 predicts the type of a location l by generating a persistent nonterminal ofcell $l\ tp$.) Then, dest/fulfill checks to make sure that the generated continuation frame has the right type relative to the destinations it connects.

Taken together, the rules dest/promise and dest/fulfill rules in Figure 9.11 create a tree-structured map of destinations starting from an initial destination d_0 and an initial persistent atomic proposition ofdest $d_0\ tp_0$, and the dest/fulfill rule ensures that every destination on this map encodes a specific and well-typed stack of frames that can be read off by following destinations back to the root d_0. The initial ofdest proposition takes over for the mobile proposition gen $tp_0\ d_0$ that formed the root of our tree in all previous specifications. The mobile gen nonterminal no longer needs indices, and just serves to place a single eval or retn somewhere on the well-typed map of destinations. The initial state of our generative traces is therefore $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph)$; this is reflected in the the preservation theorem.

Lemma (Unique indices of $\Sigma_{Gen9.11}$)**.** *Both* $\{\mathsf{ofdest}/1\}$ *and* $\{\mathsf{gendest}/1, \mathsf{cont}/2\}$ *are unique index sets under the initial state* $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph)$ *and the signature* $\Sigma_{Gen9.11}$.

Proof. Induction and case analysis on the last steps of a given trace. □

Lemma (Inversion – Figure 9.11)**.**

1. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{y{:}\langle \mathsf{eval}\ e\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ d\ tp\ e\ (z' \bullet x \bullet\ !N))$,
   where $x{:}\langle \mathsf{ofdest}\ d\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$,
   and Δ *is the persistent part of* $(\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$.

2. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{y{:}\langle \mathsf{retn}\ v\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/retn}\ d\ tp\ v\ (z' \bullet x \bullet\ !N \bullet\ !N_v))$,
   where $x{:}\langle \mathsf{ofdest}\ d\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{of}\ v\ tp\ true$, $\Psi; \Delta \vdash N : \mathsf{value}\ v\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$,
   and Δ *is the persistent part of* $(\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$.

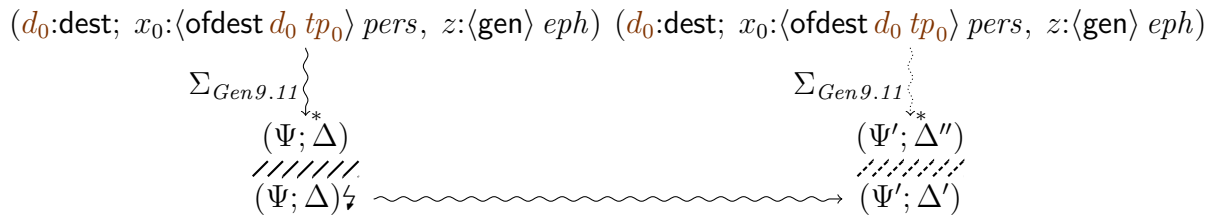
3. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph)$
$\qquad\qquad \rightsquigarrow^*_{\Sigma_{Gen9.11}}\ (\Psi;\ \Theta\{y{:}\langle \mathsf{cont}\ f\ d'\ d\rangle\ pers\})$,
then $T = (T';\{y\} \leftarrow \mathsf{dest/fulfill}\ d'\ d\ f\ tp'\ tp\ (y' \bullet\ !N \bullet x' \bullet x))$,
where $x'{:}\langle \mathsf{ofdest}\ d'\ tp'\rangle\ pers \in \Delta$, $x{:}\langle \mathsf{ofdest}\ d\ tp\rangle\ pers \in \Delta$, $\Psi;\Delta \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$
$T' :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph)$
$\qquad\qquad \rightsquigarrow^*_{\Sigma_{Gen9.11}}\ (\Psi;\ \Theta\{y'{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\})$,
and Δ *is the persistent part of* $(\Psi;\ \Theta\{z'{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\})$.

Proof. Induction and case analysis on last steps of the trace T; each case is individually quite simple because Var is always a singleton $\{y\}$. While we introduce a persistent atomic proposition cont in part 3, the step that introduces this proposition can always be rotated to the end of a trace because cont propositions cannot appear in the input interface of any step in under the generative signature $\Sigma_{Gen9.11}$. This is a specific property of $\Sigma_{Gen9.11}$, but it also follows from the definition of generative signatures (Definition 9.1), which stipulates that transitions enabled by a generative signature cannot consume or mention terminals like cont.

As an aside, z will always equal z' in parts 1 and 2, but we'll never need to rely on this fact. □

Theorem 9.7 ($\Sigma_{Gen9.11}$ is a generative invariant)**.**

If $T_1 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\Delta)$ *and* $S :: (\Psi;\Delta)\lightning \rightsquigarrow (\Psi';\Delta')$ *under the signature from Figure 7.2.4, then* $(\Psi';\Delta') = (\Psi';\Delta'')\lightning$ *for some* Δ'' *such that* $T_2 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi';\Delta'')$.



Proof. As always, the proof proceeds by enumeration, inversion, and reconstruction; the cases are all fundamentally similar to the ones we have already seen. □

9.7 On mechanization

In this chapter, we have shown that generative invariants can describe well-formedness and well-typedness properties of the full range of specifications discussed in Part II of this dissertation. We have furthermore shown that these generative invariants are a suitable basis for reasoning about type preservation in these specifications. All of these proofs have a common 3-step structure:

1. Straightforward unique index properties,
2. An inversion lemma that mimics the structure of the generative signature, and

3. A preservation theorem that proceeds by enumerating transitions, applying inversion to the given generative trace, and using the result to construct a new generative trace.

Despite the fact that the inversion lemmas in this chapter technically use induction, they do so in such a trivial way that is quite possible to imagine that inversion lemmas could be automatically synthesized from a generative signature. Unique index properties may be less straightforward to synthesize, but like termination and mode properties in Twelf, they should be entirely straightforward to verify. Only the last part of step 3, the reconstruction that happens in a preservation theorem, has the structure of a more general theorem proving task. Therefore, there is reason to hope that we can mechanize the tedious results in the results in this chapter in a framework that does much of the work of steps 1 and 2 automatically.

Bibliography

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002. 2.1, 3.3.3, 4.1, 4.1.1, 4.1.3, 4.4, 5.2, 9.1.3, 10

[CS13] Iliano Cervesato and Thierry Sans. Substructural meta-theory of a type-safe language for web programming. *Fundamenta Informaticae*, 2013. 9, 9.5

[FM12] Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012. 5.2, 9.1.3, 10

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 3.3.4, 6, 6.4.1, 6.5.1, 6.5.2, 6.5.4, 8, 6.5.5, 8.5.1, 7, 9.3.2, 9.4

[MR96] Luc Moreau and Daniel Ribbens. The semantics of pcall and fork in the presence of first-class continuations and side-effects. In *Parallel Symbolic Languages and Systems (PSLS'95)*, pages 53–77. Springer LNCS 1068, 1996. 7.2, 9.6

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 9.4

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001. 3.1, 4.1, 4.4, 4.6.1, 9.1.3

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. 9, 9.1, 9.1.1