Combining Manifest Contracts with State

Michael Greenberg
Pomona College

Manifest contracts combine the rich specifications and runtime checking of higher-order contracts \[1\] with a static type discipline. Conventional type systems prevent simple errors, like calling a boolean as a function, but manifest contracts can prevent more complex errors. For example, we could give the sqrt function the very precise type \{x:Float \mid x \geq 0\} \rightarrow \{y:Float \mid y^2 < e\}, where subset types like \{x:Float \mid x \geq 0\} refer to those floating-point numbers \(x\) such that \(x \geq 0\). Extending types with contract-like specifications in code yield powerful reasoning principles \[5,22\] and better abstractions \[14\]. Sound manifest contract systems enjoy an inversion principle, invaluable for reasoning about parameters in function bodies:

\[\vdash v : \{x:T \mid e\} \implies e[v/x] \rightarrow^* \text{true}\]

Working out the metatheory for a manifest contract semantics offers evidence that the semantics correctly checks all of its specifications.

Dimoulas et al. \[8\] introduced a latent semantics for stateful contracts (in line with the implemented behavior of Racket’s contracts \[13\] Version 6.1.1 (Ch. 8)). They check contracts on both reads and writes, carefully tracking which party is to blame.

We can adapt their operational semantics for state to a manifest setting, using casts \([T_1 \Rightarrow T_2]^I\) to dynamically move values between types (Figure \[1\] we omit a detailed semantics of casts to save space). We can just follow the types: reading a guarded location reads the location and guards the result; writing a guarded location writes a (contravariantly) guarded value. Following Wadler \[29\], we could complement the blame label for writes.

What challenges remain in proving this manifest semantics and its reasoning principles correct? And how might we address these challenges? This is work in progress.

1. Moving beyond the latent semantics

Dimoulas et al. \[8\] are primarily interested in showing that their notion of blame is coherent. In papering over deeper semantic questions, they miss three interesting interactions: contracts over entire reference cells, atomic updates of separate state, and contracts on pure values with hidden state.

In Figure \[1\] we interpret Dimoulas et al.’s ref/c(e) contract as a reference containing a value of subset type \(\{x_1:T \mid e_1\}\). We could also have subset types over state: \(\{x_1;\text{Ref } T \mid e_2\}\). Such general refinements \[5\] add expressivity: \(e_1\) can only see the value stored in \(x_1\), while \(e_2\) can read and write the reference \(x_2\) itself.

Suppose we have two reference cells holding integers, \(x\) and \(y\), and \(x\)’s sign must always be the inverse of \(y\)’s. To maintain that invariant, the program must initialize and update the two values atomically. In simple cases it may suffice to instead have a ref-

---

1 These are also known as refinement types and predicate contracts \[15\].

2 See Greenberg et al. \[17\] semantics for a fuller discussion of the difference between latent and manifest contracts.
2.2 State introduces circularity
Not only can we use state to encode the usual fixpoint operators, we can use guarded locations themselves to write diverging computations:

\[
\begin{align*}
&\text{let } f : \text{Ref (Int \to Int)} = \text{ref } \lambda x. x \text{ in} \\
&\text{let } g = (\ldots \text{Ref } \{ x : \text{Int \to Int} \mid x = 0 \}) \theta f \text{ in} \\
&g := \lambda x. g := (\lambda x. x + 1 - 1); x;
\end{align*}
\]

The call to \( g \) triggers a contract check, which writes to \( g \), which triggers another check, and so on. We may want to statically forbid diverging circular contracts, but even converging circularity is an issue. Contracts can invalidate themselves:

\[
\begin{align*}
&\text{let } x : \text{Ref Int} = \text{ref } 0 \text{ in} \\
&\langle \text{Ref Int} \Rightarrow \text{Ref } \{ z : \text{Int } \mid x := -1; z \geq 0 \} \rangle \stackrel{\text{Ref Int}}{\Rightarrow} \langle x \rangle
\end{align*}
\]

The first time we check the contract, it will succeed... but not the second! What inversion principle should apply here? That is, what does the type \( \text{Ref } \{ z : \text{Int } \mid x := -1; z \geq 0 \} \) mean?

2.3 State is metatheoretically challenging
Soundness proofs for manifest systems typically involve some kind of semantic subtyping. Early work included semantic subtyping as an optimization, but also to account for the types of constants and congruence steps. (See Belo et al. [4] for a full discussion.) Proving type soundness calls for heavyweight methods, like axiomatization [12], logical relations [16], and bisimulation [22]. State complicates these proof techniques, but we can start by using step-indexing and Kripke models in our logical relations [2, 3].

3. Ways forward
How can we assign meaning to stateful contracts? Two approaches seem promising for handling scoping and circularity.

3.1 Type and effect systems
Talpin and Jouvelot [27]'s type and effect systems extend a static typing discipline with information about effects. Some of this information will be critical for scoping. In order to have nonReentrant from Section 1 have a well formed dependent type, we must account for the allocation that happens in its body:

\[
\forall \alpha. \beta. (\alpha \to \beta) \text{ inside } \{ x : \alpha \not\text{inside} \to \beta \}
\]

Not only will effect annotations resolve issues with scoping, they’ll be essential for exposing the stateful parts invariants—the contract might also want to track when inside is true.

Effect tracking could help us manage circularity in contracts. Effect-free contracts aren’t circular. We can allow some effects while still avoiding circularity by putting memory locations (or other possible sources of effects) in a partial order. Contracts written at level \( l \) can allow effects at lower levels \( l' \subset l \).

To manage scoping, we’ll need static effect tracking. But dynamic or gradual (mixed static and dynamic) approaches would suffice [4] for dealing with circularity.

3.2 Information flow control
Information flow control (IFC) offers a strong security guarantee: noninterference. That is, changing the secure inputs to a program doesn’t change the insecure outputs. IFC won’t help with scoping, but it can prevent circularities in contract checking. We can mark contract code as insecure and mark some bits of state as secure—then contracts won’t interfere (read or write) such forbidden state.

IFC can be enforced statically, dynamically, or gradually [9, 21, 24]. Either way would be fine for enforcing non-circularity, but how do we take a noninterference proof and use it to derive inversion principles and type soundness?

4. Other related work
Xi and Pfenning [30]'s subset types use a more restricted language of indices than Ou et al. [18]; Rondon et al. [20] makes a similar restriction on the language of indices. Svendsen et al. [25], F∗ [26] and Casinghino et al. [6]’s \( X^\theta \) mix dependently typed programs with effects, but checking dependent and other ‘refined’ types statically—which we check dynamically. Ahman [11] studies a static refinement type theory over an algebraic theory of effects.

Tov and Pucella [28] use stateful latent contracts to dynamically check interactions between affine and unlimited types. Ideally, a manifest version of their makeAffineFunContract could be proved correct directly using the reasoning principles of a stateful manifest language (an open version [31] of a parametricity relation [5]).

Owens [19] interprets contract checking itself as an effect, following Degen et al. [17]’s observations about contracts and laziness.

Shinnar [23] uses delimited checkpoints to implement stateful latent contracts in Haskell. Checking is delimited—contracts roll back their writes after checking. Can we adapt these techniques to manifest contracts, which need not just an intuitive semantics, but a coherent notion of meaning?

Acknowledgments
Greg Morrisett suggested the idea of using noninterference to avoid circularity in contract checking. This work was supported in part by the NSF under grant CNS 1111520 at Princeton University.

References


