Combining Manifest Contracts with State

Michael Greenberg

Pomona College

Manifest contracts combine the rich specifications and runtime checking of higher-order contracts [11] 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 $\{x{:}\mathsf{Float} \mid x \geq 0\} \rightarrow \{y{:}\mathsf{Float} \mid |x^2 - y| < \epsilon\}$, where *subset types* like $\{x{:}\mathsf{Float} \mid x \geq 0\}$ refer to those floating-point numbers x such that $x \geq 0$.[1] 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:

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

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*[2] 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* $\langle T_1 \Rightarrow T_2 \rangle^l$ 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 $\mathsf{ref/c}(\kappa)$ contract as a reference containing a value of subset type ($\mathsf{Ref}\ \{x_1{:}T \mid e_1\}$). We could also have subset types over state: $\{x_1{:}\mathsf{Ref}\ T \mid e_2\}$. 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 reference cell holding a pair, but in general this won't be possible: stateful contract languages need atomic updates.

[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.

Syntax extensions for state

$$\begin{array}{lcll} e & ::= & \ldots \mid \mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 & \textit{terms} \\ v & ::= & \ldots \mid \gamma & \textit{values} \\ \gamma & ::= & \mathsf{loc} \mid \langle \mathsf{Ref}\ T_1 \Rightarrow \mathsf{Ref}\ T_2 \rangle^l\ \gamma & \textit{guarded locations} \\ \sigma & : & \mathsf{Locations} \rightharpoonup \mathsf{Values} & \textit{stores} \end{array}$$

Operational semantics for state $\boxed{\sigma, e_1 \longrightarrow \sigma, e_2}$

$$\begin{array}{r} \sigma, \mathsf{ref}\ v \longrightarrow \sigma \uplus [\mathsf{loc} \mapsto v], \mathsf{loc} \\ \sigma, !\mathsf{loc} \longrightarrow \sigma, \sigma(\mathsf{loc}) \\ \sigma, \mathsf{loc} := v \longrightarrow \sigma[\mathsf{loc} \mapsto v], \mathsf{unit} \\ \sigma, !(\langle \mathsf{Ref}\ T_1 \Rightarrow \mathsf{Ref}\ T_2 \rangle^l\ \mathsf{loc}) \longrightarrow \sigma, \langle T_1 \Rightarrow T_2 \rangle^l\ !\mathsf{loc} \\ \sigma, (\langle \mathsf{Ref}\ T_1 \Rightarrow \mathsf{Ref}\ T_2 \rangle^l\ \mathsf{loc}) := v \longrightarrow \sigma, \mathsf{loc} := \langle T_2 \Rightarrow T_1 \rangle^{\bar{l}}\ v \end{array}$$

Figure 1. Operational semantics for manifest contracts with state

We can construct a contract over a seemingly pure type that is nevertheless stateful. Such contracts are *temporal contracts* [10]—their meaning varies over time. For example, we can take a function and wrap it to ensure that it is not called during its own dynamic extent:

$$\begin{array}{l} \mathsf{let\ nonReentrant} = \\ \quad \Lambda\alpha\beta.\ \lambda f : (\alpha \rightarrow \beta).\ \mathsf{let\ inside} = \mathsf{ref\ false\ in} \\ \quad\quad \lambda\{x{:}\alpha \mid \mathsf{not}\ !\mathsf{inside}\}. \\ \quad\quad\quad \mathsf{inside} := \mathsf{true}; \mathsf{let}\ y = f\,x\ \mathsf{in\ inside} := \mathsf{false}; y \end{array}$$

2. Challenges

State and effects complicate type theories with subset types. Ou et al. [18] were the first to combine code-based specifications and references. They use special *dynamic reference cells* to mediate interactions between simple and dependent code, with a packing/unpacking discipline corresponding closely to Dimoulas et al.'s guarded locations (though Ou et al. don't use blame).

Ou et al. try to bake in optimizations: they augment their calculus with a theorem prover that identifies checks that always succeed and can be omitted. But they pay a price, restricting the set of terms that can appear as predicates in subset types.

We set a more modest goal: define a sound, stateful manifest contract calculus with arbitrary predicates—we can optimize after the fact [5, 22]. We highlight three problems before discussing possible solutions in Section 3.

2.1 State complicates scoping

Earlier work on dependent manifest contracts resolved dependency with substitution, but that won't work when we add state. What type does $\mathsf{nonReentrant}$ from Section 1 have? It must be:

$$\forall\alpha\beta.\ (\alpha \rightarrow \beta) \rightarrow \{x{:}\alpha \mid \mathsf{not}\ !\mathsf{inside}\} \rightarrow \beta$$

Unfortunately, this type isn't well formed; inside isn't in scope.

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 contracts:

$$\begin{array}{l} \mathsf{let}\ f : \mathsf{Ref}\ (\mathsf{Int} \to \mathsf{Int}) = \mathsf{ref}\ \lambda x.\ x\ \mathsf{in} \\ \mathsf{let}\ g = \langle \ldots \Rightarrow \mathsf{Ref}\ \{x{:}\mathsf{Int} \to \mathsf{Int} \mid x\ 0 = 0\}\rangle^l\ f\ \mathsf{in} \\ \quad g := \lambda x.\ g := (\lambda x.\ x + 1 - 1); x; \\ \quad !g\ 0 \end{array}$$

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 divergently circular contracts, but even converging circularity is an issue. Contracts can invalidate themselves:

$$\begin{array}{l} \mathsf{let}\ x : \mathsf{Ref\ Int} = \mathsf{ref}\ 0\ \mathsf{in} \\ \quad \langle \mathsf{Ref\ Int} \Rightarrow \mathsf{Ref}\ \{z{:}\mathsf{Int} \mid x := -1; z \geq 0\}\rangle^l\ x \end{array}$$

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 $\mathsf{Ref}\ \{z{:}\mathsf{Int} \mid x := -1; z \geq 0\}$ *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. [5] for a full discussion.) Proving type soundness calls for heavyweight methods, like axiomatization [12, 18], 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:

$$\forall \alpha\beta.\ (\alpha \to \beta) \overset{\mathsf{alloc\ inside}}{\to} \{x{:}\alpha \mid \mathsf{not\ !inside}\} \to \beta$$

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 $l' \sqsubset l$.

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 λ^θ mix dependently typed programs with effects, but checking dependent and other 'refined' types statically—which we check dynamically. Ahman [1] 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. [7]'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

[1] D. Ahman. Refinement types and algebraic effects, 2013. HOPE.

[2] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Principles of Programming Languages (POPL)*, 2009. doi: 10.1145/1480881.1480925.

[3] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, Sept. 2001. doi: 10.1145/504709.504712.

[4] F. Bañados Schwerter, R. Garcia, and E. Tanter. A theory of gradual effect systems. In *International Conference on Functional Programming (ICFP)*, 2014. doi: 10.1145/2628136.2628149.

[5] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *European Symposium on Programming (ESOP)*, 2011.

[6] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *Principles of Programming Languages (POPL)*, 2014. doi: 10.1145/2535838.2535883.

[7] M. Degen, P. Thiemann, and S. Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *ATPS*, Lübeck, Germany, 2009.

[8] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *Programming Languages and Systems*, volume 7211. 2012. doi: 10.1007/978-3-642-28869-2_11.

[9] T. Disney and C. Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

[10] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *International Conference on Functional Programming (ICFP)*. ACM, 2011. doi: 10.1145/2034773.2034800.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[12] C. Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, 2006.

[13] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[14] M. Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, November 2013.

[15] M. Greenberg. A refinement type by any other name, Mar. 2015. URL `http://goo.gl/KaHLBG`. Blog post.

[16] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL)*, 2010.

[17] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. *Journal of Functional Programming (JFP)*, 22(3):225–274, May 2012.

[18] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science (TCS)*, 2004.

[19] Z. Owens. Contract monitoring as an effect, 2012. HOPE.

[20] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Principles of Programming Languages (POPL)*, 2010.

[21] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5 – 19, jan 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121.

[22] T. Sekiyama, A. Igarashi, and M. Greenberg. Polymorphic manifest contracts, revised and resolved, 2015. In submission.

[23] A. Shinnar. *Safe and Effective Contracts*. PhD thesis, Harvard University, May 2011.

[24] D. Stefan, A. Russo, J. Mitchell, and D. Maziéres. Flexible dynamic information flow control in haskell. In *Haskell Symposium*, 2011.

[25] K. Svendsen, L. Birkedal, and A. Nanevski. Partiality, state and dependent types. In *Typed Lambda Calculi and Applications*, volume 6690. 2011. doi: 10.1007/978-3-642-21691-6_17.

[26] N. Swamy, C. Hriţcu, C. Keller, P.-Y. Strub, A. Rastogi, A. Delignat-Lavaud, K. Bhargavan, and C. Fournet. Semantic purity and effects reunited in F*. Unpublished., 2015. URL `https://www.fstar-lang.org/papers/icfp2015/`.

[27] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, June 1994. doi: 10.1006/inco.1994.1046.

[28] J. Tov and R. Pucella. Stateful contracts for affine types. In *Programming Languages and Systems*, volume 6012 of *LNCS*. 2010. doi: 10.1007/978-3-642-11957-6_29.

[29] P. Wadler. A Complement to Blame. In *SNAPL*, volume 32 of *LIPIcs*, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.309.

[30] H. Xi and F. Pfenning. Dependent types in practical programming. In *Principles of Programming Languages (POPL)*, 1999. doi: 10.1145/292540.292560.

[31] J. Zhao, Q. Zhang, and S. Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In *Programming Languages and Systems*, volume 6461 of *LNCS*. 2010. doi: 10.1007/978-3-642-17164-2_24.