Phantom Contracts for Better Linking

Daniel Patterson — Northeastern University — dbp@ccs.neu.edu

POPL 2019 *Student Research Competition* Extended Abstract

Introduction There are dozens of programming languages in wide use, and people are designing new ones every day. Increasingly, languages provide programmers with rich types to enforce invariants about their code statically. But an outstanding challenge remains how to integrate components written in new languages into extant systems. Programmers currently rely on unsafe FFIs that defer errors to runtime (if errors are reported at all), failing both to permit a gradual evolution from older languages to a newer ones, and to support graceful integration of domain specific languages.

The essential problem is that, at the point of linking (in low-level bytecode, object code, etc), any static type information that could have been used to report errors has already been discarded. To support linking errors, e.g., that a function is called with the wrong number or type of arguments, we must *preserve* type information. But what should target type systems look like? One option is to add *just enough* types to support what you are compiling. In that case, as soon as you build a compiler from a sufficiently different source language, you will have to extend this target language. Another option is to try to give the target language the richest possible type system: e.g., a dependent high-order separation logic ([5] or similar), increasing the complexity of implementing compilers.

We propose a third option: enrich an existing possibly-typed target language with **phantom contracts**, which are written in a rich but simple operational language, but that *run at type-checking time*. The phantom language can then be used to encode source type information that cannot be captured in the existing target and to enforce flexible linking invariants. Phantom contracts are refinements to target types that operate over ghost state used in type checking. Source types translate to target phantom computations that, after linking and type checking, are erased.

By Example We demonstrate the idea by example. We show compilers from two source languages into a target with phantom contracts. We contrast this with the typical approach for type-preserving compilation, using this to show how phantom contracts work and why they are an improvement.

We begin with our source languages. One is Index$^{\pm}$, the simply-typed lambda calculus with integers enriched with an positive/negative index algebra that can be used to track the sign of numbers. The syntax and a selection of the typing rules are in Figure 1; operational semantics are standard, not depending on indexes. Number types η capture our index system, where literal numbers are either $+\mathtt{0}$ or $-$, and computations can degrade to the unknown $?$. Number type polymorphism allows functions like $\mathtt{e} = \Lambda\alpha.\lambda\mathtt{x}{:}\alpha.\mathtt{x} * \mathtt{2}$, which has type $\forall\alpha.\alpha * +\mathtt{0}$. Then $\mathtt{e}[+\mathtt{0}]\mathtt{2}$ would have type $+\mathtt{0}$, since type application reduces $+\mathtt{0} * +\mathtt{0}$ to $+\mathtt{0}$.

Our second source language is BabyDill, a simply typed linear lambda calculus based on removing polymorphism from PDILL[13] (and adding a base type). The syntax and a few typing rules are shown in Figure 2 (the reader is encouraged to consult [13] for more details).

Phantom Contracts A typical approach to supporting type-preserving compilation from various languages is to incorporate all necessary features directly into the target. We would therefore end up with a target with sign indexes and linearity (at least), and we would have some translation into this target. If BabyDill had integers, we could compile literal integers to precise types, but any source integer types would translate to whatever $?$ translated to, to capture the fact that BabyDill does not track sign. Here the compiler can determine how to translate representations—there also may be cases where we need programmer annotations in order to figure out what code to generate, which is an idea we have explored in previous work called **linking types** [7].

This approach, while theoretically straightforward, has a major downside: it's likely that the target will be specialized to the particular source language(s) chosen, and as a result would have to change in order to support more languages. Over time, it may converge to something rich enough to capture almost everything needed, but at that point the language would be a hodge-podge of features.

Rather than try to incorporate all of the type features of the source languages into the type system of the target, we propose enriching the target language with **phantom contracts**, that on their own have straightforward semantics but can be used by compiler

$$
\begin{array}{llll}
\textsc{Index}^{\pm} & \tau & ::= & \eta \mid \forall\alpha.\tau \mid \tau \to \tau \\
& \eta & ::= & \alpha \mid +\mathtt{0} \mid - \mid\ ? \mid \eta{+}\eta \mid \eta{*}\eta \\
& \mathtt{e} & ::= & \mathtt{n} \mid \mathtt{x} \mid \mathtt{e} + \mathtt{e} \mid \mathtt{e} * \mathtt{e} \mid \lambda\mathtt{x}:\tau.\,\mathtt{e} \mid \mathtt{e}\,\mathtt{e} \\
& & & \Lambda\alpha.\mathtt{e} \mid \mathtt{e}[\eta] \\
& \mathtt{v} & ::= & \mathtt{n} \mid \lambda\mathtt{x}:\tau.\,\mathtt{e} \mid \Lambda\alpha.\mathtt{e}
\end{array}
$$

$$
\frac{\mathtt{n} \geq 0}{\mathrm{H};\Gamma \vdash \mathtt{n} : +\mathtt{0}} \qquad \frac{\mathtt{n} < 0}{\mathrm{H};\Gamma \vdash \mathtt{n} : -} \qquad \frac{\mathtt{x} : \tau \in \Gamma}{\mathrm{H};\Gamma \vdash \mathtt{x} : \tau}
$$

$$
\frac{\mathrm{H};\Gamma \vdash \mathtt{e}_1 : \eta_1 \qquad \mathrm{H};\Gamma \vdash \mathtt{e}_2 : \eta_2}{\mathrm{H};\Gamma \vdash \mathtt{e}_1 * \mathtt{e}_2 : \Downarrow(\eta_1 * \eta_2)} \qquad \frac{\mathrm{H},\alpha;\Gamma \vdash \mathtt{e} : \tau}{\mathrm{H};\Gamma \vdash \Lambda\alpha.\mathtt{e} : \forall\alpha.\tau}
$$

$$
\frac{\mathrm{H};\Gamma \vdash \mathtt{e} : \forall\alpha.\tau}{\mathrm{H};\Gamma \vdash \mathtt{e}[\eta] : \Downarrow(\tau[\eta/\alpha])}
$$

$$
\begin{array}{lcl}
\Downarrow(+\mathtt{0} + +\mathtt{0}) & = & +\mathtt{0} \\
\Downarrow(- + +\mathtt{0}) & = & ? \\
\Downarrow(?{+}\eta) & = & ? \\
\mathtt{etc}\ldots & & \\
\Downarrow(\eta_1{+}\eta_2) & = & \eta_1{+}\eta_2
\end{array}
$$

Figure 1: Syntax and (very) selected static semantics for Index$^{\pm}$.

$$
\begin{array}{llll}
\textsc{BabyDill} & \tau & ::= & \mathtt{unit} \mid \tau \multimap \tau \mid !\tau \mid \tau\&\tau \mid \tau \otimes \tau \\
& \mathtt{e} & ::= & () \mid \mathtt{x} \mid \mathtt{a} \mid \lambda\mathtt{a} : \tau.\mathtt{e} \mid \mathtt{e}\ \mathtt{e}' \mid !\mathtt{e} \\
& & & \mathtt{let}\ !\mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e}' \mid \langle \mathtt{e}, \mathtt{e}' \rangle \mid \mathtt{e}.1 \mid \mathtt{e}.2 \\
& & & (\mathtt{e}, \mathtt{e}) \mid \mathtt{let}\ (\mathtt{a}, \mathtt{a}') = \mathtt{e}\ \mathtt{in}\ \mathtt{e}' \\
& \mathtt{v} & ::= & () \mid \lambda\mathtt{a} : \tau.\mathtt{e} \mid !\mathtt{e} \mid \langle \mathtt{e}, \mathtt{e}' \rangle \mid (\mathtt{v}, \mathtt{v}')
\end{array}
$$

$$
\frac{}{\Gamma;\mathtt{a} : \tau \vdash \mathtt{a} : \tau} \qquad \frac{\mathtt{x} : \tau \in \Gamma}{\Gamma;\cdot \vdash \mathtt{x} : \tau} \qquad \frac{\Gamma;\Delta,\mathtt{a} : \tau_1 \vdash \mathtt{e} : \tau_2}{\Gamma;\Delta \vdash \lambda\mathtt{a} : \tau_1.\mathtt{e} : \tau_1 \multimap \tau_2}
$$

$$
\frac{\Gamma;\Delta_1 \vdash \mathtt{e}_1 : \tau_1 \multimap \tau_2 \qquad \Gamma;\Delta_2 \vdash \mathtt{e}_2 : \tau_1 \qquad \Delta_1,\Delta_2 = \Delta}{\Gamma;\Delta \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau_2}
$$

$$
\frac{\Gamma;\Delta \vdash \mathtt{e}_1 : \tau_1 \qquad \Gamma;\Delta \vdash \mathtt{e}_2 : \tau_2}{\Gamma;\Delta \vdash \langle \mathtt{e}_1, \mathtt{e}_2 \rangle : \tau_1\&\tau_2} \qquad \frac{\Gamma;\cdot \vdash \mathtt{e} : \tau}{\Gamma;\cdot \vdash \mathtt{e} : !\tau}
$$

Figure 2: Syntax and (very) selected static semantics for BabyDill.

writers to encode complex invariants. In Figure 3 we present syntax and a small selection of the static semantics for PhantomLC, which is the simply typed lambda calculus with pairs and recursive types extended with phantom contracts. We highlight a few important details, and then consider our two-language case study.

First, note that our *static* semantics involve *running* our phantom contracts. Our phantom language is a simply typed lambda calculus with sexpressions, a first-order store, and `assert`ions that can fail (there is no reduction rule for $\delta^1(\texttt{assert}, \texttt{false})$). The phantom computations are guaranteed to terminate or fail, and these two possibilities indicate whether the invariants encoded in the phantom contracts were satisfied. This allows compilers to encode their representations of types and the computational fragments that are necessary for checking that the fragments that they receive have the proper phantom representations.

PhantomLC

$$
\begin{array}{rrcl}
 & \tau & ::= & \texttt{int} \mid \tau \to \tau' \mid \mu\alpha.\tau \mid \alpha \\
 & \texttt{e} & ::= & \hat{\texttt{e}}\{\varphi\} \\
 & \hat{\texttt{e}} & ::= & \texttt{x} \mid \texttt{n} \mid \texttt{e} + \texttt{e} \mid \texttt{e} * \texttt{e} \mid \lambda\texttt{x}{:}\tau.\texttt{e} \mid \texttt{e}\ \texttt{e}' \\
 & & & (\texttt{e}, \texttt{e}) \mid \texttt{fst}\,\texttt{e} \mid \texttt{snd}\,\texttt{e} \mid \texttt{fold}\,\texttt{e} \\
 & & & \texttt{unfold}\,\texttt{e} \mid \texttt{let}\ \texttt{x}\{\nu\}{:}\tau = \texttt{e}\ \texttt{in}\ \texttt{e}' \\
 & \texttt{v} & ::= & \texttt{n} \mid \texttt{fold}\,\texttt{v} \mid \lambda\texttt{x}{:}\tau.\texttt{e} \mid (\texttt{v}, \texttt{v}) \\
 & \varphi & ::= & \texttt{sexp} \mid \texttt{ref}\,\texttt{sexp} \mid \varphi := \varphi \mid !\varphi \mid \nu \\
 & & & \texttt{match}\,\varphi \mid \texttt{n}\nu_1.\varphi_1 \mid \texttt{s}\nu_2.\varphi_2 \mid \texttt{b}\nu_3.\varphi_3 \mid (\nu_4, \nu_5).\varphi_4 \\
 & & & \lambda\nu{:}\varphi\tau.\varphi \mid \varphi\,\varphi' \mid \varphi; \varphi' \mid \delta^{\texttt{n}}(\texttt{op}^{\texttt{n}}, \varphi_1, \ldots \varphi_{\texttt{n}}) \\
\texttt{sexp} & ::= & \texttt{n} \mid \texttt{s} \mid \texttt{true} \mid \texttt{false} \mid (\texttt{sexp}, \texttt{sexp}') \\
\texttt{op}^1 & ::= & \texttt{assert} \mid \texttt{not} \mid \texttt{length} \mid \ldots \\
\texttt{op}^2 & ::= & \texttt{sexp}{=} \mid \texttt{append} \mid + \mid * \mid \ldots \\
 & \varphi\tau & ::= & \texttt{sexp} \mid \texttt{ref}\,\texttt{sexp} \mid \varphi\tau \to \varphi\tau
\end{array}
$$

$$\frac{\Gamma; \texttt{S} \vdash \hat{\texttt{e}} : \tau; \texttt{S}' \qquad \Gamma \vdash (\texttt{S}', \varphi) \Downarrow (\texttt{S}'', \varphi')}{\Gamma; \texttt{S} \vdash \hat{\texttt{e}}\{\varphi\} : \tau; \texttt{S}''}$$

$$\frac{\Gamma; \texttt{S}_1 \vdash \hat{\texttt{e}}_1 : \tau; \texttt{S}_2 \qquad \Gamma \vdash (\texttt{S}_2, \varphi) \Downarrow (\texttt{S}_3, \varphi') \qquad \Gamma, \texttt{x}{:}\tau, \nu{:}\varphi'; \texttt{S}_3 \vdash \texttt{e}_2{:}\tau_2; \texttt{S}_4}{\Gamma; \texttt{S}_1 \vdash \texttt{let}\ \texttt{x}\{\nu\}{:}\tau = \hat{\texttt{e}}_1\{\varphi\}\ \texttt{in}\ \texttt{e}_2 : \tau_2; \texttt{S}_4}$$

$$\frac{\nu{:}\varphi \in \Gamma}{\Gamma \vdash (\texttt{S}, \nu) \Downarrow (\texttt{S}, \varphi)} \qquad \frac{\Gamma \vdash (\texttt{S}, \varphi) \Downarrow (\texttt{S}', \texttt{true})}{\Gamma \vdash (\texttt{S}, \delta^1(\texttt{assert}, \varphi)) \Downarrow (\texttt{S}', \texttt{true})}$$

$$\frac{\Gamma \vdash (\texttt{S}, \varphi_1) \Downarrow (\texttt{S}', \texttt{s}) \qquad \Gamma, \nu{:}\texttt{s} \vdash (\texttt{S}', \varphi_2) \Downarrow (\texttt{S}'', \varphi_2')}{\Gamma \vdash (\texttt{S}, \texttt{match}\ \varphi_1 \ldots \mid \texttt{s}\nu.\varphi_2 \ldots)}$$

Figure 3: Syntax and selected static semantics for PhantomLC.

Compilers using Phantom Contracts To compile Index$^{\pm}$, we consider translating well-typed expressions. This means our compiler will be defined case-wise over the typing rules of the language. We show an illustrative portion of it in Figure 4, where $(\texttt{e})^+$ is the compiled term. The key case in the figure, when considering linking, is application. If the argument came from another language, compiler, etc, we could use the same target code, replacing $\texttt{e}'^+$ with that code. We can see that the rest of the compiler would then be able to enforce the sign restrictions from the source, because they were encoded by the compiler into the phantom contracts and checked at that point.

To compile BabyDill, we again define a translation case-wise over typing derivations. A portion of this translation is shown in Figure 5. Note that the only thing we use our phantom contracts to track is that linearity is not violated, by using reference counts in our phantom store. This presentation is slightly simplified, as the full translation needs to implement a caller-save strategy on the reference counts to support variable shadowing, which we elide for space. One case that is important is lazy (i.e., additive) pairs (of

$$
\begin{array}{rcl}
(\texttt{e}_1 + \texttt{e}_2 : \eta)^+ & = & \texttt{let}\ \_\texttt{e}_1\{\nu_1\} = \texttt{e}_1{}^+\ \texttt{in} \\
 & & \texttt{let}\ \_\texttt{e}_2\{\nu_2\} = \texttt{e}_2{}^+\ \texttt{in} \\
 & & (\_\texttt{e}_1 + \_\texttt{e}_2)\{\langle\!\langle\eta\rangle\!\rangle\} \\
(\texttt{x} : \tau)^+ & = & \texttt{x}\{\delta^1(\texttt{assert}, \delta^2(\texttt{sexp}{=}, \langle\!\langle\tau\rangle\!\rangle, \nu_{\texttt{x}}))\; ; \langle\!\langle\tau\rangle\!\rangle\} \\
(\lambda\texttt{x}{:}\tau.\texttt{e} : \tau \to \tau')^+ & = & (\lambda\texttt{x}{:}\tau^+.\texttt{let}\ \_\{\nu_{\texttt{x}}\} = \texttt{0}\{\langle\!\langle\tau\rangle\!\rangle\}\ \texttt{in}\ \texttt{e}^+) \\
 & & \{\langle\!\langle\tau \to \tau'\rangle\!\rangle\} \\
(\texttt{e}\ \texttt{e}' : \tau))^+ & = & \texttt{let}\ \texttt{f}\{\nu_{\texttt{f}}\} = \texttt{e}^+\ \texttt{in} \\
 & & \texttt{let}\ \texttt{a}\{\nu_{\texttt{a}}\} = \texttt{e}'^+\ \texttt{in} \\
 & & (\texttt{f}\ \texttt{a})\{\texttt{match}\ \nu_{\texttt{f}} \\
 & & \mid (\prime\mapsto\prime, (\nu_{\tau'}, \nu_\tau)). \\
 & & \delta^1(\texttt{assert}, \delta^2(\texttt{sexp}{=}, \nu_{\tau'}, \nu_{\texttt{a}}))\; ; \langle\!\langle\tau\rangle\!\rangle\} \\
\langle\!\langle\tau \to \tau'\rangle\!\rangle & = & (\prime\mapsto\prime, (\langle\!\langle\tau\rangle\!\rangle, \langle\!\langle\tau'\rangle\!\rangle)) \qquad ?^+ = \texttt{int} \\
\langle\!\langle\texttt{+0}\rangle\!\rangle & = & \prime + \texttt{0}\prime \qquad\qquad +\texttt{0}^+ = \texttt{int}
\end{array}
$$

Figure 4: Selections from Index$^{\pm}$ to PhantomLC compiler.

type $\tau_1 \& \tau_2$), for which only one projection is used. In that case, we run the phantom contracts on one side, reset reference counts, and then run the other, so that both sides use all variables. The same technique would also be used to support `if`. Other cases have empty phantom contracts, because we only have to handle cases that affect the reference counts.

$$
\begin{array}{rcl}
(\lambda\texttt{a} : \tau_1.\texttt{e} : \tau_1 \multimap \tau_2)^+ & = & \lambda\texttt{a} : \tau_1{}^+.\texttt{let}\_\{\texttt{a}\} = \texttt{ref}\ 1\ \texttt{in} \\
 & & \texttt{let}\_\texttt{r}\{\_\texttt{r}\} = \texttt{e}^+\ \texttt{in} \\
 & & \_\texttt{r}\{\delta^1(\texttt{assert}, \delta^2(\texttt{int}\,{=}, !\texttt{a}, \texttt{0}))\; ; \_\texttt{r}\} \\
(\texttt{a} : \tau)^+ & = & \texttt{a}\{\texttt{a} := \delta^2(-, !\texttt{a}, 1)\} \\
(\Gamma; \texttt{a}_1 : \tau_1 \ldots \vdash & = & \texttt{let}\ \texttt{v}_1\{\} = \texttt{e}_1{}^+\ \texttt{in}\ \texttt{let}\_\{\} = \texttt{0}\{\delta^1(\texttt{assert}, \\
\langle\texttt{e}_1, \texttt{e}_2\rangle : \tau_1 \& \tau_2)^+ & & \delta^2(\texttt{int}\,{=}, !\texttt{a}_1, \texttt{0}))\ldots; \texttt{a}_1 := 1\ldots\}\ \texttt{in}\ (\texttt{v}_1, \texttt{e}_2{}^+)
\end{array}
$$

Figure 5: Selections from BabyDill to PhantomLC compiler.

As before, we can see what linking we can support. With no shared base types, the default interactions are pretty limited, but again if we extended BabyDill with integers, we could extend our compiler to support them. But in this case, the compilation is more interesting, because what we would compile them to is the particular choice of phantom contracts used by the Index$^{\pm}$ compiler, rather than to a built-in feature of the target language. The resulting interoperability would be the same, and indeed, that's the goal: phantom contracts are not about making fundamentally easier the task of building a given type-preserving compiler, but rather to prevent the issue of having to migrate the entire ecosystem because some feature needs to be added to the target language. With such a system, type-preserving compilers could be developed independently, and only at the point of interface do common types and calling conventions need to be established, similarly to how they have to be established with existing FFIs. Individual compiler writers could experiment with different representations, benefiting from not having to change the underlying target language infrastructure, and getting baseline interoperability to the extent that they provide phantom interoperability wrappers.

Related Work The name should hint this derives inspiration both from phantom types ([1]) and contracts (e.g., soft contract verification [6]) / gradual typing, where the notion of encoding typing constraints operationally is familiar (e.g., AGT [3]), though the fact that our operational semantics is at type-checking time makes things quite different. But we also draw much inspiration from refinement type systems (e.g., [2, 8, 12], where the last made an explicit connection to static higher-order contracts) and from ghost state, both from separation logic (e.g., [4]) and F*'s ghost monad [11] (and earlier ghost refinements [10]). We also see connections to work translating typing rules into rewrite systems (e.g., [9]).

References

[1] Matthew Fluet and Riccardo Pucella. 2002. Phantom Types and Subtyping. In Proceedings of the Second IFIP International Conference on Theoretical Computer Science (TCS '02).

[2] Tim Freeman. 1994. Refinement Types for ML. PhD Thesis.

[3] Ron Garcia, Alison Clark, Éric Tanter. 2016 Abstracting Gradual Typing. In 43rd ACM Symposium on Principles of Programming Languages (POPL '16).

[4] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In 21st ACM International Conference on Functional Programming (ICFP '16).

[5] Aleksander Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. 2007. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In 16th European Symposium on Programming (ESOP '07).

[6] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft contract verification. In 19th ACM International Conference on Functional Programming (ICFP '14).

[7] Daniel Patteron and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In SNAPL: Summit on Advances in Programming Languages (SNAPL'17), May 2017.

[8] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid Types. In 30th ACM Conference on Programming Language Design and Implementation (PLDI '08).

[9] Aaron Stump, Garrin Kimmell, Roba El Haj Omar. 2011. Type Preservation as a Confluence Problem. In Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11).

[10] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, Jean Yang. Secure Distributed Programming with Value-dependent Types. 2011. In 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11).

[11] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In 43rd ACM Symposium on Principles of Programming Languages (POPL '16).

[12] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In 22nd European Symposium on Programming (ESOP '13).

[13] Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. 2010. Relational Parametricity for a Polymorphic Linear Lambda Calculus. In Proceedings of the 8th Asian conference on Programming languages and systems (APLAS'10).