Compiler Correctness via Contextual Equivalence

Matthew McKay, advised by Karl Crary

Abstract

We have developed a methodology for verifying the correctness of the closure conversion phase of a compiler, adapted from the work by Perconti and Ahmed. This lets us verify that individual components of programs are compiled correctly, so they can be linked with any other code and still behave as desired. We do this by using a combined language that encompasses both the source and target languages in which the compiled code can be reasoned about alongside its source, which we do using contextual equivalence. Our main improvement over previous methods is that we do not need boundaries that separate the source and target language while inside the combined language.

Combined Language

Our language C is a combined language of the source and target languages, with all terms and types from both languages. The source language S is System F extended with recursive functions and existentials. The target T is the closure converted version of the source, so it is the same except it has closed recursive functions and application instead of normal recursive functions and application.

$$\begin{aligned}
\tau &::= \alpha \mid \mathtt{unit} \mid \mathtt{int} \mid \tau \times \tau \mid \tau \to \tau \mid \tau \Rightarrow \tau \mid \forall\alpha.\tau \mid \exists\alpha.\tau \\
e &::= () \mid n \mid e\,\mathtt{p}\,e \mid \mathtt{ifz}(e,e,e) \mid x \mid \langle e,e\rangle \mid \pi_i e \\
&\quad \mid \mathtt{fun}\ f(x:\tau).e \mid e\ e \mid \overline{\mathtt{fun}}\ f(x:\tau).e \mid e \wedge e \\
&\quad \mid \Lambda\alpha.e \mid e[\tau] \mid \mathtt{pack}[\tau', e]\,\mathtt{as}\,\exists\alpha.\tau \mid \mathtt{unpack}[\alpha, x] = e\,\mathtt{in}\,e \\
\mathtt{p} &::= + \mid - \mid * \\
\Gamma &::= \cdot \mid \Gamma, x:\tau \\
\Delta &::= \cdot \mid \Delta, \alpha
\end{aligned}$$

Two Different Functions

The main thing of note about the combined language is the presence of two types of recursive functions. There are normal recursive functions, and then there are closed recursive functions, which are generated from normal recursive functions during closure conversion. They behave the same as normal recursive functions, however they can only use their argument and bound function variable, all other variables are not in scope inside the function (hence "closed").

$$\frac{\Delta \vdash \tau\,\mathtt{type} \quad \Delta;\Gamma, f:\tau\to\tau', x:\tau \vdash e:\tau'}{\Delta;\Gamma \vdash \mathtt{fun}\ f(x:\tau).e : \tau \to \tau'}\ \mathit{Tfun}$$

$$\frac{\Delta \vdash \tau\,\mathtt{type} \quad \Delta;f:\tau\to\tau', x:\tau \vdash e:\tau'}{\Delta;\Gamma \vdash \overline{\mathtt{fun}}\ f(x:\tau).e : \tau \Rightarrow \tau'}\ \mathit{Tccfun}$$

Closure Conversion

The closure conversion translation is essentially the standard translation, with the slight difference that the functions in the translated language are denoted as closed recursive functions. Thus we have the type transaltion for functions as

$$|\tau_1 \to \tau_2| = \exists\alpha.((|\tau_1| \times \alpha) \Rightarrow |\tau_2|) \times \alpha$$

For closure conversion the only interesting translation rules are for functions and applications, the rules for which are below:

$$\frac{\begin{array}{c}\Gamma = x_1:\tau_1,...,x_n:\tau_n \qquad \Delta \vdash_S \tau\,\mathtt{type} \\ \Delta;\Gamma, x:\tau, f:\tau\to\tau' \vdash_S e:\tau' \rightsquigarrow \bar{e} \quad \tau_{env} = |\tau_1| \times ... \times |\tau_n|\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash_S \mathtt{fun}\ f(x:\tau).e : \tau\to\tau' \rightsquigarrow \mathtt{pack}[\tau_{env}, \\ \langle(\overline{\mathtt{fun}}\ f(y:|\tau| \times \tau_{env}).[F/f]S(\bar{e})), E\rangle]\,\mathtt{as}\,|\tau\to\tau'|\end{array}}\ \mathit{Rfun}$$

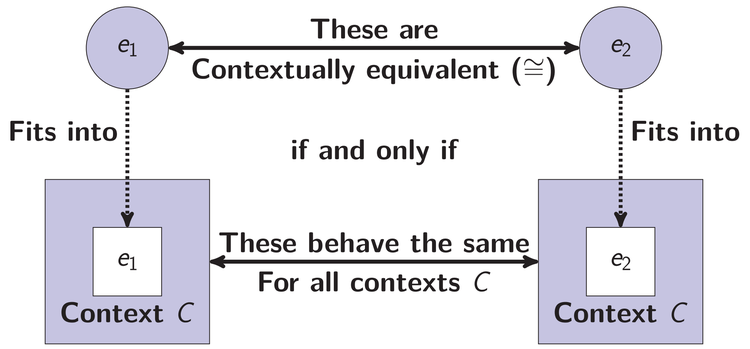
$$\frac{\Delta;\Gamma \vdash_S e_1 : \tau\to\tau' \rightsquigarrow \bar{e_1} \quad \Delta;\Gamma \vdash_S e_2 : \tau \rightsquigarrow \bar{e_2}}{\Delta;\Gamma \vdash_S e_1\ e_2 : \tau' \rightsquigarrow \mathtt{unpack}[\alpha,x] = \bar{e_1}\,\mathtt{in}(\pi_1 x) \wedge \langle \bar{e_2}, \pi_2 x\rangle}\ \mathit{Rapp}$$

The first rule uses the following definitions:
$E = \langle x_1, \langle ...\langle x_{n-1}, x_n\rangle ...\rangle\rangle$
$S = [\pi_1 y/x][\pi_1\pi_2 y/x_1]...[\pi_1\pi_2...\pi_2 y/x_{n-1}][\pi_2...\pi_2 y/x_n]$
$F = \mathtt{pack}[\tau_{env}, \langle f, \pi_2 y\rangle]\,\mathtt{as}\,|\tau\to\tau'|$

Contextual Equivalence

The idea behind contextual equivalence is that for two terms to be equivalent, any program that could use them will behave the same no matter which term is used inside it. Formally, we write $\Delta;\Gamma \vdash_C e_1 \cong e_2 : \tau$ if both terms e_1 and e_2 can be typed at τ, and for every context $\mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\cdot \triangleright \mathtt{int})$ (a program with a hole in it which both e_1 and e_2 fit into), we have that $\mathcal{C}\{e_1\} \simeq \mathcal{C}\{e_2\}$, which simply states that $\mathcal{C}\{e_1\}$ and $\mathcal{C}\{e_2\}$ will have the same terminating behavior (either loop or terminate).

Contextual Equivalence Visualization



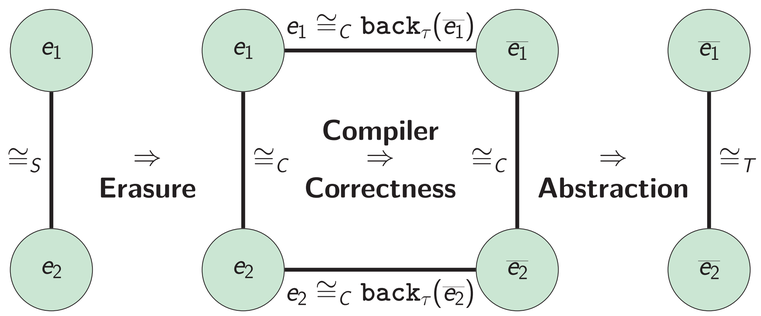
Statement of Compiler Correctness Theorem

Given a term $\Delta;\Gamma \vdash_S e_1 : \tau$ whose translation is $\Delta;\Gamma \vdash_S e_1 : \tau \rightsquigarrow \overline{e_1}$, we have that

$$\Delta;\Gamma \vdash e_1 \vdash \mathtt{back}_\tau([\mathtt{over}_\Gamma/\Gamma]\overline{e_1}) : \tau$$

This essentially says that e_1 is contextually equivalent to $\overline{e_1}$, however these two terms don't have the same type in the combined language. Thus we need to wrap the compiled $\overline{e_1}$ in an $\mathtt{over}_\Gamma$ substitution and $\mathtt{back}_\tau$ to make it have a matching type. Thus this theorem guarantees that the compiler preserves the behavior of the code that it compiles, so we have a verification of its correctness.

Equivalence Preservation Visualization



The $\mathtt{over}_\tau$ substitutions for compiler correctness are left out for conciseness.

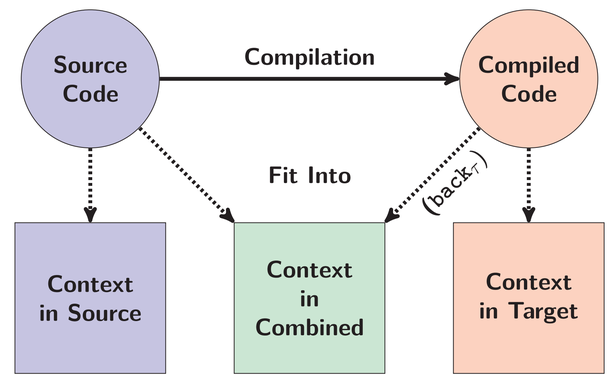
Statement of Equivalence Preservation Theorem

Given terms $\Delta;\Gamma \vdash_S e_1 : \tau$ and $\Delta;\Gamma \vdash_S e_2 : \tau$ whose translations are $\Delta;\Gamma \vdash_S e_1 : \tau \rightsquigarrow \overline{e_1}$ and $\Delta;\Gamma \vdash_S e_2 : \tau \rightsquigarrow \overline{e_2}$, then we have that

$$\Delta;\Gamma \vdash_S e_1 \cong e_2 : \tau \Rightarrow \Delta;\Gamma \vdash_T \overline{e_1} \cong \overline{e_2} : \tau$$

This essentially states that if two terms are equivalent in the source language, then the terms that they compile to are equivalent in the target language. Therefore, the compiler preserves equivalence.
To prove this we use our compiler correctness theorem, but we also need to show that if two terms are equivalent in S then they are equivalent in C, and similarly if two terms are equivalent in C then they are equivalent in T. The latter is simple since T is a subset of C, but for the former we need another translation, which we call erasure.

Combined Language Context Visualization



Over and Back Functions

We define two functions in the combined language, $\mathtt{over}_\tau : \tau \to |\tau|$ and $\mathtt{back}_\tau : |\tau| \to \tau$, each of which take terms in either the source or the target language and changes them at the top level to have the correct converted type in the opposite language. The two functions are mutually recursive and are inverses of one another.

$$\begin{aligned}
\mathtt{over}_\alpha &= \lambda x:\alpha.x \\
\mathtt{over}_{\mathtt{unit}} &= \lambda x:\mathtt{unit}.x \\
\mathtt{over}_{\mathtt{int}} &= \lambda x:\mathtt{int}.x \\
\mathtt{over}_{\tau_1\times\tau_2} &= \lambda x:\tau_1\times\tau_2.\langle \mathtt{over}_{\tau_1}\pi_1 x, \mathtt{over}_{\tau_2}\pi_2 x\rangle \\
\mathtt{over}_{\tau_1\to\tau_2} &= \lambda f:\tau_1\to\tau_2.\,\mathtt{pack}[\tau_1\to\tau_2, \langle \overline{\lambda y:|\tau_1| \times (\tau_1\to\tau_2)}. \\
&\qquad \mathtt{over}_{\tau_2}((\pi_2 y)\ (\mathtt{back}_{\tau_1}\pi_1 y)), f\rangle]\,\mathtt{as}\,|\tau_1\to\tau_2| \\
\mathtt{over}_{\forall\alpha.\tau} &= \lambda x:(\forall\alpha.\tau).\Lambda\alpha.(\mathtt{over}_\tau(x[\alpha])) \\
\mathtt{over}_{\exists\alpha.\tau} &= \lambda x:(\exists\alpha.\tau).\,\mathtt{unpack}[\alpha,y] = x\,\mathtt{in}(\mathtt{pack}[\alpha, \mathtt{over}_\tau(y)]\,\mathtt{as}\,|\exists\alpha.\tau|) \\
\hline
\mathtt{back}_\alpha &= \lambda x:\alpha.x \\
\mathtt{back}_{\mathtt{unit}} &= \lambda x:\mathtt{unit}.x \\
\mathtt{back}_{\mathtt{int}} &= \lambda x:\mathtt{int}.x \\
\mathtt{back}_{\tau_1\times\tau_2} &= \lambda x:|\tau_1\times\tau_2|.\langle \mathtt{back}_{\tau_1}\pi_1 x, \mathtt{back}_{\tau_2}\pi_2 x\rangle \\
\mathtt{back}_{\tau_1\to\tau_2} &= \lambda f:|\tau_1\to\tau_2|.\lambda y:\tau_1.\,\mathtt{unpack}[\alpha,g] = f\,\mathtt{in} \\
&\qquad \mathtt{back}_{\tau_2}((\pi_1 g) \wedge \langle \mathtt{over}_{\tau_1} y, \pi_2 g\rangle) \\
\mathtt{back}_{\forall\alpha.\tau} &= \lambda x:|\forall\alpha.\tau|.\Lambda\alpha.(\mathtt{back}_\tau(x[\alpha])) \\
\mathtt{back}_{\exists\alpha.\tau} &= \lambda x:|\exists\alpha.\tau|.\,\mathtt{unpack}[\alpha,y] = x\,\mathtt{in}(\mathtt{pack}[\alpha, \mathtt{back}_\tau(y)]\,\mathtt{as}\,\exists\alpha.\tau)
\end{aligned}$$

Erasure

The erasure translation e° simply removes closed recursive functions by converting them to normal recursive functions. So $(\tau_1\to\tau_2)^\circ = \tau_1^\circ \to \tau_2^\circ$ and

$$(\overline{\mathtt{fun}}\ f(x:\tau).e)^\circ = \mathtt{fun}\ f(x:\tau^\circ).e^\circ$$

The point of erasure is to say that adding closed recursive functions to S doesn't give it any more capability than it already has, which makes sense as closed functions do exactly the same thing as normal functions. This can then be used to show that equivalent terms in S are equivalent in C, since removing the closed functions doesn't affect the language's power.

Future Work

There is plenty more work that can be extended from this research, as it was mostly a demonstration of a possible approach. The most logical extension would be to apply this methodology to other translation phases of a compiler, like the allocation phase. Unfortunately other phases will be significantly more complicated to do, as the combined language increases in complexity as the difference between the source and target languages increases. Closure conversion is a good starting place due to the similarity between the languages, though it could be possible to extend this methodology to other phases of a compiler.