Dependent-Type-Preserving Memory Allocation

PAULETTE KORONKEVICH[*], University of British Columbia, Canada, pletrec@cs.ubc.ca

1 INTRODUCTION

Dependently typed programming languages such as Coq, Agda, Idris, and F*, allow programmers to write detailed specifications of their programs and prove their programs meet these specifications. However, these specifications can be violated during compilation since they are erased after type checking. External programs linked with the compiled program can violate the specifications of the original program and change the behavior of the compiled program—even when compiled with a verified compiler. For example, since Coq does not allow explicitly allocating memory, a programmer might link their Coq program with a C program that can allocate memory. Even if the Coq program is compiled with a verified compiler like CertiCoq [1], the external C program can still violate the memory-safe specification of the Coq program by providing an uninitialized pointer to memory. This error could be ruled out by type checking in a language expressive enough to indicate whether memory is initialized versus uninitialized. Linking with a program with an uninitialized pointer could be considered ill-typed, and our linking process could prevent linking with ill-typed programs. To facilitate type checking during linking, we can use *type-preserving compilation*, which preserves the types through the compilation process.

In this ongoing work, we develop a typed intermediate language (IL) CC-CC$_A$ that supports dependent memory allocation, as well as a dependent-type-preserving compiler pass for memory allocation. The dependent-type-preserving compiler pass allocates heap space for dependent pairs and closures from the CC-CC language developed by Bowman and Ahmed [3]. This work is a significant step towards developing a full dependent-type-preserving compiler. In particular, combining this pass with previous work on two major passes, CPS [4] and closure conversion [3], a dependent-type-preserving compiler to a low-level language like C is possible.

2 TYPED MEMORY ALLOCATION

The source language CC-CC of this translation is based on the Calculus of Constructions (CC); however, first-class functions are replaced by closed code and closures. CC-CC includes one impredicative universe $\star$, and one predicative universe $\square$. Expressions have no explicit distinction between terms, types, or kinds, but we use the meta-variable e to evoke a term expression and A or B to evoke a type expression. The syntax of expressions include a unit value $\langle\rangle$ and its type $\mathsf{1}$, let expressions $\mathsf{let}\,\mathsf{x} = \mathsf{e} : \mathsf{A}\,\mathsf{in}\,\mathsf{e}'$, closed code $\lambda\,(\mathsf{n} : \mathsf{A}', \mathsf{x} : \mathsf{A}).\,\mathsf{e}$ and dependent code types $\mathsf{Code}\,(\mathsf{n} : \mathsf{A}', \mathsf{x} : \mathsf{A}).\,\mathsf{B}$, closure values $\langle\langle \mathsf{e}, \mathsf{e}\rangle\rangle\,\mathsf{as}\,\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ and dependent closure types $\Pi\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, application $\mathsf{e}\,\mathsf{e}'$ (which applies closures instead of functions), dependent pairs $\langle \mathsf{e}, \mathsf{e}'\rangle\,\mathsf{as}\,\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$ and their type $\Sigma\,\mathsf{x} : \mathsf{A}.\,\mathsf{B}$, and finally first and second projection, $\mathsf{fst}\,\mathsf{e}$ and $\mathsf{snd}\,\mathsf{e}$. The typing, subtyping, and conversion rules for CC-CC are the same as given by Bowman and Ahmed [3].

The target language CC-CC$_A$ includes all the features of CC-CC, extending the syntax with an explicit dependent memory allocation $\mathbf{malloc}_{\mathbf{x}}\,[\mathbf{A}, \mathbf{B}]$, and memory initialization operators $\mathbf{e}[1]\longleftarrow\mathbf{e}'$ and $\mathbf{e}[2]\longleftarrow\mathbf{e}'$. CC-CC$_A$ only allocates tuples of two machine words; this suffices for our translation, but it should not be difficult to extend the language to arbitrary length tuples. The type of dependent pairs $\Sigma\,\mathbf{x} : \mathbf{A}^{\phi}.\,\mathbf{B}^{\phi}$ includes an initialization flag ϕ to indicate whether the elements have been initialized (1) or not (0), as is done by Morrisett et al. [5]. This prevents the first and second projection from accessing unallocated memory. Finally, since closures are allocated as pairs, we use a tag $\mathbf{Clos}\,\mathbf{e}$ to indicate the pair should be treated as a closure.

[*] Graduate (Ph.D.) student advised by William J. Bowman, ACM number: 3780425

$$\boxed{\Psi;\Gamma \vdash e : A}$$

$$\text{[Clo]}\quad \frac{\Psi;\Gamma \vdash e : \Sigma\, y : (\mathsf{Code}\,(x_1 : A_1, x : A).\, B)^1 .\, A_1^1}{\Psi;\Gamma \vdash \mathsf{Clos}\, e : \Pi\, x : A[\mathsf{snd}\, e/x_1].\, B[\mathsf{snd}\, e/x_1]}
\qquad
\text{[Malloc]}\quad \frac{\Psi;\Gamma \vdash A : U \qquad \Psi;\Gamma, x : A \vdash B : U}{\Psi;\Gamma \vdash \mathsf{malloc}_x\,[A, B] : \Sigma\, x : A^0.\, B^0}$$

$$\text{[Assign1]}\quad \frac{\Psi;\Gamma \vdash e : \Sigma\, x : A^0.\, B^{\phi} \qquad \Psi;\Gamma \vdash e' : A}{\Psi;\Gamma \vdash e[1] \leftarrow e' : \Sigma\, x : A^1.\, B^{\phi}}
\qquad
\text{[Assign2]}\quad \frac{\Psi;\Gamma \vdash e : \Sigma\, x : A^1.\, B^0 \qquad \Psi;\Gamma \vdash e' : B[\mathsf{fst}\, e/x]}{\Psi;\Gamma \vdash e[2] \leftarrow e' : \Sigma\, x : A^1.\, B^1}$$

$$\text{[Fst]}\quad \frac{\Psi;\Gamma \vdash e : \Sigma\, x : A^1.\, B^{\phi}}{\Psi;\Gamma \vdash \mathsf{fst}\, e : A}
\qquad
\text{[Snd]}\quad \frac{\Psi;\Gamma \vdash e : \Sigma\, x : A^1.\, B^1}{\Psi;\Gamma \vdash \mathsf{snd}\, e : B[\mathsf{fst}\, e/x]}$$

Fig. 1. CC-CC$_A$ Typing (excerpt)

$$\boxed{\llbracket e \rrbracket = e}$$

$$\begin{array}{rcl}
\llbracket \Sigma\, x : A.\, B \rrbracket & \stackrel{\text{def}}{=} & \Sigma\, x : \llbracket A \rrbracket^1 .\, \llbracket B \rrbracket^1 \\
\llbracket \langle e_1, e_2 \rangle \text{ as } \Sigma\, x : A.\, B \rrbracket & \stackrel{\text{def}}{=} & \mathsf{let}\; y = \mathsf{malloc}_x\,[\llbracket A \rrbracket, \llbracket B \rrbracket] : \Sigma\, x : \llbracket A \rrbracket^0 .\, \llbracket B \rrbracket^0 \;\mathsf{in} \\
 & & \mathsf{let}\; y_1 = y[1] \leftarrow \llbracket e_1 \rrbracket : \Sigma\, x : \llbracket A \rrbracket^1 .\, \llbracket B \rrbracket^0 \;\mathsf{in} \\
 & & \mathsf{let}\; y_2 = y_1[2] \leftarrow \llbracket e_2 \rrbracket : \Sigma\, x : \llbracket A \rrbracket^1 .\, \llbracket B \rrbracket^1 \;\mathsf{in} \\
 & & y_2 \\
\llbracket \langle\!\langle e_1, e_2 \rangle\!\rangle \text{ as } \Pi\, x : A.\, B \rrbracket & \stackrel{\text{def}}{=} & \mathsf{let}\; y = \mathsf{malloc}_x\,[\llbracket \mathsf{Code}\,(x' : A_1, x : A).\, B \rrbracket, \llbracket A_1 \rrbracket] \;\mathsf{in} \\
 & & \mathsf{let}\; y_1 = y[1] \leftarrow \llbracket e_1 \rrbracket : \Sigma\, x : \llbracket \mathsf{Code}\,(x' : A_1, x : A).\, B \rrbracket^1 .\, A_1^0 \;\mathsf{in} \\
 & & \mathsf{let}\; y_2 = y_1[2] \leftarrow \llbracket e_2 \rrbracket : \Sigma\, x : \llbracket \mathsf{Code}\,(x' : A_1, x : A).\, B \rrbracket^1 .\, A_1^1 \;\mathsf{in} \\
 & & \mathsf{Clos}\; y_2
\end{array}$$

Fig. 2. Allocation Translation (excerpt)

The typing rules for CC-CC$_A$ remain mostly the same as CC-CC; however, we now include the heap while type checking to access tuples that have been allocated. The heap Ψ consists of locations with their types $\ell : A$ as well as locations mapped to heap values $\ell \mapsto \langle e_1, e_2 \rangle$. Small step reduction $\triangleright$ and conversion $\triangleright^*$ are defined over configurations $\langle \Psi \mid e \rangle$ to access the allocated tuples. For example, first and second projection of a pair are defined as follows:

$$\begin{array}{rcll}
\langle \Psi \mid \mathsf{fst}\, \ell \rangle & \triangleright_{\pi_1} & \langle \Psi \mid e_1 \rangle & \text{where } \ell \mapsto \langle e_1, e_2 \rangle \in \Psi \\
\langle \Psi \mid \mathsf{snd}\, \ell \rangle & \triangleright_{\pi_2} & \langle \Psi \mid e_2 \rangle & \text{where } \ell \mapsto \langle e_1, e_2 \rangle \in \Psi
\end{array}$$

The typing rules for memory allocation, initialization, closure tagging, and first and second projection are given in Figure 1. Memory allocation, as expected, types as a dependent pair with the initialization flags set to 0. Memory initialization changes the flag on the dependent pair type depending on which element is initialized. Finally, [Clo] types a dependent pair as a Π type, given the elements of the pair are closed code and its environment.

The essence of the translation is given in Figure 2. All other cases simply recursively translate subexpressions. For pairs and closures, the translated program first initializes memory according to the type of the dependent pair (or the closure). Then, the program initializes each element of the pair with the result of translating the subexpressions e_1 and e_2. Finally, it returns the result of this allocation, tagging it as a closure in the closure case.

3 TYPE PRESERVATION

Type preservation guarantees that a well-typed source program is compiled to a well-typed target program, and is stated as follows:

THEOREM 3.1. *If* $\Gamma \vdash \mathsf{e} : \mathsf{A}$ *then* $\cdot; [\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] : [\![\mathsf{A}]\!]$.

Since typing in dependently typed languages relies on subtyping and equivalence, and equivalence relies on conversion, we must also prove that subtyping, equivalence, and conversion are preserved by the allocation pass:

LEMMA 3.2. *If* $\Gamma \vdash \mathsf{A} \preceq \mathsf{B}$ *then* $\cdot; [\![\Gamma]\!] \vdash [\![\mathsf{A}]\!] \preceq [\![\mathsf{B}]\!]$.

LEMMA 3.3. *If* $\Gamma \vdash \mathsf{e} \equiv \mathsf{e}'$ *then* $\cdot; [\![\Gamma]\!] \vdash [\![\mathsf{e}]\!] \equiv [\![\mathsf{e}']\!]$.

LEMMA 3.4. *If* $\Gamma \vdash \mathsf{e} \rhd \mathsf{e}'$ *then* $\cdot; [\![\Gamma]\!] \vdash \langle \cdot \mid [\![\mathsf{e}]\!] \rangle \rhd \langle \Psi \mid [\![\mathsf{e}']\!] \rangle$.

Finally, since many dependent typing rules substitute terms into types (such as [SND]), we must also show that substitution is preserved:

LEMMA 3.5. $[\![\mathsf{e}[\mathsf{e}'/\mathsf{x}]]\!] \equiv [\![\mathsf{e}]\!][[\![\mathsf{e}']\!]/\mathbf{x}]$.

The proofs of all these lemmas and the main type preservation theorem follow by straightforward induction over the source derivation. Intuitively, the translation is only adding series of **let** statements which explicitly allocate a pair (or closure). The final compiled program should be a pair (or closure) of the same type.

4 FUTURE WORK

Proving type preservation to an arbitrary target language is not enough, as the target language itself should be type-safe and consistent (that is, we cannot prove false). We prove CC-CC$_A$ to be type-safe and consistent by providing a model in extensional CIC (eCIC). We then prove the model preserves typing and the definition of the empty type ($\bot$), following a technique well explained by Boulier et al. [2]. If CC-CC$_A$ was inconsistent, then we could produce a proof of false $\bot$ and translate it to $\bot$ in eCIC; however, since no such proof exists in eCIC, CC-CC$_A$ must be consistent.

We translate each dependent pair to a dependent pair in eCIC, where each element has type Maybe A, with a proof about which element of the pair is filled based on the initialization flags:

$$\begin{aligned}
[\![\Sigma\,\mathbf{x}:\mathbf{A}^0.\,\mathbf{A}^0]\!]_M &\overset{\text{def}}{=} \Sigma\, p : (\Sigma\, x : \text{Maybe }[\![\mathbf{A}]\!]_M.\,\text{Maybe }[\![\mathbf{B}]\!]_M).\, p = \langle \text{None}, \text{None}\rangle \\
[\![\Sigma\,\mathbf{x}:\mathbf{A}^1.\,\mathbf{A}^0]\!]_M &\overset{\text{def}}{=} \Sigma\, p : (\Sigma\, x : \text{Maybe }[\![\mathbf{A}]\!]_M.\,\text{Maybe }[\![\mathbf{B}]\!]_M).\, p = \langle \text{Just } e, \text{None}\rangle \\
[\![\Sigma\,\mathbf{x}:\mathbf{A}^1.\,\mathbf{A}^1]\!]_M &\overset{\text{def}}{=} \Sigma\, p : (\Sigma\, x : \text{Maybe }[\![\mathbf{A}]\!]_M.\,\text{Maybe }[\![\mathbf{B}]\!]_M).\, p = \langle \text{Just } e_1, \text{Just } e_2\rangle
\end{aligned}$$

For readibility we exclude the existential quantification for each element inside Just expressions. We then add auxiliary definitions to eCIC, maybe-snd and maybe-fst, which are guaranteed to produce the first and second element of a pair, since they expect a pair alongside a proof that the elements are filled:

$$\begin{aligned}
&\text{maybe-fst} : \Sigma\, p : (\Sigma\, x : \text{Maybe } A.\,\text{Maybe } B).\, p = \langle \text{Just } e, \_\rangle \longrightarrow A \\
&\text{maybe-snd} : \Sigma\, p : (\Sigma\, x : \text{Maybe } A.\,\text{Maybe } B).\, p = \langle \text{Just } e_1, \text{Just } e_2\rangle \longrightarrow B[e_1/x]
\end{aligned}$$

Any instances of **fst** and **snd** are then translated to maybe-fst and maybe-snd. Based on [FST] and [SND], the translation of **e** should have the type expected by these definitions, so the model should be type-preserving. Proving this translation to eCIC type preserving is still ongoing.

An exciting extension of CC-CC$_A$ would be an explicit **free** operator to deallocate memory. This IL with **free** would be an interesting language to implement a garbage collector for the dependent-type-preserving compiler model so far, giving us even further confidence in the correct evaluation of dependently-typed programs. This IL would likely need richer specifications of memory operations, like those provided by Hoare Type Theory [6].

REFERENCES

[1] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *International Workshop on Coq for Programming Languages (CoqPL).* http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf

[2] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Conference on Certified Programs and Proofs (CPP).* https://doi.org/10.1145/3018610.3018620

[3] William J. Bowman and Amal Ahmed. 2018. Typed Closure Conversion for the Calculus of Constructions. In *International Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3192366.3192372

[4] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-preserving CPS Translation of Σ and Π Types Is Not Not Possible. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL (Jan. 2018). https://doi.org/10.1145/3158110

[5] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (May 1999). https://doi.org/10.1145/319301.319345

[6] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. In *International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/1159803.1159812