A Trace Model of an Imperative Multi-Stage Language

Haoxuan Yin[1], Andrzej Murawski[1], and Luke Ong[2]

[1]University of Oxford
[2]Nanyang Technological University

Multi-stage programming languages allow users to write programs that can be executed at separate stages. They provide a mechanism of converting a general-purpose program into a specialized, more efficient program. For example, consider the imperative power function given by Calcagno, Moggi, and Sheard (2003). The program is written in modern MetaOCaml syntax (Kiselyov, 2023; Kiselyov, 2014) for readability.

```
let rec power n x y =
    if n = 0 then y := 1
    else begin power (n-1) x y; y := !y * x end
let rec power_staged_gen n x y =
    if n = 0 then .< .~y := 1 >.
    else .< .~(power_staged_gen (n - 1) x y); .~y := !(.~y) * .~x >.
let power_staged n = Runcode.run
    .< fun x y -> .~(power_staged_gen n .<x>. .<y>.) >.
let power_staged_3 = power_staged 3
(* power_staged_3 = fun x y ->
    begin y := 1; y := !y * x; y := !y * x; y := !y * x end *)
```

The *power* n x y function is an unstaged, general-purpose program that computes x^n and stores the result in y. Sometimes, we may know the value of n before knowing the values of x and y, or we may need to run the program multiple times for a fixed n and various different xs and ys. In these cases, it is helpful to carry out the computations related to n in advance. The *power_staged* function is the same imperative program as *power* except for staging annotations and η expansions. It treats n as a normal variable that is known immediately, and x and y as staged variables that will only be supplied later. Then, for example, we can generate the function *power_staged_3* as soon as we have a particular $n = 3$. This specialized function is more efficient than the original *power* function because it does not contain recursion or branching.

MetaML (Taha and Sheard, 2000) is the classic language for multi-stage programming. It is theoretically the same as and practically the ancestor of modern MetaOCaml. As shown in the above example, MetaML has three key constructs. *Bracket* $\langle M \rangle$ turns a term into a piece of code. A piece of code is considered a value and its content is not evaluated during execution. *Escape* $\tilde{}M$ removes the outermost bracket in M and is used to assemble small pieces of code into a larger one. *Run* **run** M executes the content of the code M.

When staging a conventional program, a critical concern is whether the staging is faithful. Of course, we do not expect the staged program to have exactly the same meaning as the original program, because staging can advance or postpone certain computations. But we do expect them to be equivalent under some additional conditions. In the above imperative power example, a reasonable expectation is that *power* and $\lambda nxy.power\_staged\ n\ x\ y$ should be equivalent. To address this concern, we need to settle a theoretical foundation for the following question:

When are two imperative MetaML programs equivalent?

This is essentially the *full abstraction* problem (Milner, 1977), where one looks for a sound and complete model that captures *contextual equivalence*. Two programs M_1 and M_2 are said to be contextually equivalent if their observable behaviours such as termination are the same in all suitable contexts.

Operational game semantics (Laird, 2007; Jaber and Murawski, 2021) has been successfully used to construct fully abstract models for various programming languages. It represents the behaviour of a program in a context as a sequence of interactions—called a *trace*—between the program and the context. The semantics of a program is then given by a *trace model*, which consists of all the traces that the program can generate according to a set of transition rules. This approach is valued for its intuitive, operational nature and is particularly well-suited to imperative programming languages.

In this paper, we consider an imperative variant of MetaML, $\text{MiniML}^{\text{meta}}_{\text{ref}}$, as introduced by Calcagno, Moggi, and Sheard (2003), and give a sound and complete operational game semantics for this language. Our main contributions are as follows.

1 Partially Closed Instances of Use

Programming language researchers utilize *closed instances of use* (CIU) (Honsell et al., 1995; Talcott, 1998) approximations, which say that equivalence under all suitable contexts coincides with equivalence under all suitable evaluation contexts, heaps, and substitutions that close all free variables. Inoue and Taha (2016) pointed out that the evaluation of a closed term in MetaML might involve evaluation of open terms in the process. In contrast to conventional programs, where the body of an abstraction $\lambda x.M$ is never evaluated until x has been substituted away by β-reduction, in staged languages evaluation can *go under a lambda*, if the lambda abstraction is deferred to a later stage and the redex is fired at the current stage. For example, the closed term $\langle \lambda x.\tilde{}((\lambda y.y)\langle x\rangle)\rangle$ can be decomposed into evaluation context $\langle \lambda x.\tilde{}\bullet\rangle$ and redex $(\lambda y.y)\langle x\rangle$, and the redex contains a free occurrence of the variable x. Traditional proof methods for CIU fail because of this.[1] In this paper, we define a *partially closed instances of use* (PCIU) approximation instead, and show that it coincides with contextual equivalence for $\text{MiniML}_{\text{ref}}^{\text{meta}}$. As evaluations can only go under a lambda abstraction when it occurs at a future stage (stage 1 or above), in PCIU we only consider substitutions that substitute away all variables declared to be at the current stage (stage 0). And the resulting term is *partially closed*, because it may only contain free variables at stage 1 or above.

2 Trace Model

With the above preparations, we give a fully abstract trace model for $\text{MiniML}_{\text{ref}}^{\text{meta}}$. As an example, consider the following two traces that might be generated by the imperative power programs above.

$$\mathtt{t}_1 = (\overline{f_1}, \emptyset)\ (f_1(3), \emptyset)\ (\overline{f_2}, \emptyset)\ (f_2(2), \emptyset)\ (\overline{f_3}, \emptyset)\ (f_3(\ell_1), \{\ell_1 \mapsto 0\})\ (\overline{8}, \{\ell_1 \mapsto 8\})$$
$$\mathtt{t}_2 = (\overline{f_1}, \emptyset)\ (f_1(-1), \emptyset)\ (\overline{f_2}, \emptyset)$$

One of the traces that can be generated by *power_staged* is $\mathtt{t}_1$. In this trace, the program first announces itself as a function represented by the name f_1. Then the context asks the result of applying the function f_1 to 3, and the program responds with another function name f_2. The name f_2 essentially denotes the result of computing *power_staged*_3, but its underlying value is hidden to the context. The context further supplies two arguments $x = 2$ and $y = \ell_1$, and in the end the program can respond that the result of the computation is 8 and modify y accordingly.

The trace $\mathtt{t}_1$ can also be generated by *power* and $\lambda nxy.power\_staged\ n\ x\ y$, indicating that all three programs can compute the power function correctly. However, as a direct consequence of staging, *power_staged* starts computation as soon as n is available, while the other two only start computation when all three arguments are available. Therefore, when we supply a faulty $n = -1$, *power* and $\lambda nxy.power\_staged\ n\ x\ y$ will still be able to produce trace $\mathtt{t}_2$, while *power_staged* fails to do so because it enters a loop immediately. Our model will be able to give

$$\text{Tr}(power\_staged) \subsetneq \text{Tr}(power) = \text{Tr}(\lambda nxy.power\_staged\ n\ x\ y)$$

and given that it is fully abstract, we can deduce

$$power\_staged \lneq power \approx \lambda nxy.power\_staged\ n\ x\ y$$

in terms of contextual equivalence. The equivalence $power \approx \lambda nxy.power\_staged\ n\ x\ y$ shows the faithfulness of staging in the sense that it never gives wrong answers.

3 Applications

Now we use examples to illustrate how our system can be used to prove equivalences or inequivalences between programs. For a start, we observe that $\text{MiniML}_{\text{ref}}^{\text{meta}}$ turns out be a conservative extension of the underlying stage-free language $\text{MiniML}_{\text{ref}}$ (Calcagno, Moggi, and Sheard, 2003).

Then we explore the relationship among MetaML operators. The operator $\langle M\rangle$ turns a term of type t into a term of type $\langle t\rangle$, while the operators $\tilde{}M$ and **run** M turn a term of type $\langle t\rangle$ into a term of type t. Therefore, intuitively, the latter two operators can be thought of as inverses of the former. For example, we have $\langle 1+1\rangle \approx \langle\tilde{}\langle 1+1\rangle\rangle \approx \langle \textbf{run}\ \langle 1+1\rangle\rangle$. Indeed, when inserted into $C = \textbf{if}\ (\textbf{run}\ \bullet = 2)\ ()\ \Omega$, where Ω is some divergent term, all three terms terminate. We use our trace model to prove new equational rules that make this intuition precise.

Then we use the imperative power function to illustrate how we can formally prove that staging is faithful. We also give a general theorem stating the condition under which staging is correct.

Finally, a natural question to ask is whether there exist general equivalences between staged and unstaged programs in the style of an erasure theorem (Inoue and Taha, 2016). We give a negative answer to this question by giving terms that only differ in staging annotations but evaluate to different values. This is because staging can change the execution order of statements, which alters the semantics of a program in an imperative language.

[1] We have not been able to prove the CIU theorem or to give a counterexample to it in $\text{MiniML}_{\text{ref}}^{\text{meta}}$.

References

Calcagno, Cristiano, Eugenio Moggi, and Tim Sheard (May 2003). "Closed Types for a Safe Imperative MetaML". In: *Journal of Functional Programming* 13.3, pp. 545–571. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796802004598. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/closed-types-for-a-safe-imperative-metaml/155E678C81DAE8E2E945180BE177D414 (visited on 02/16/2024).

Honsell, Furio et al. (1995). "A Variable Typed Logic of Effects". In: *Inf. Comput.* 119.1, pp. 55–90. DOI: 10.1006/INCO.1995.1077. URL: https://doi.org/10.1006/inco.1995.1077 (visited on 03/14/2025).

Inoue, Jun and Walid Taha (Jan. 2016). "Reasoning about Multi-Stage Programs". In: *Journal of Functional Programming* 26, e22. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796816000253. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/reasoning-about-multistage-programs/60E3E040633DD97C4B61766123F1D639 (visited on 02/16/2024).

Jaber, Guilhem and Andrzej S. Murawski (2021). "Complete Trace Models of State and Control". In: *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Nobuko Yoshida. Vol. 12648. Lecture Notes in Computer Science. Springer, pp. 348–374. DOI: 10.1007/978-3-030-72019-3_13. URL: https://doi.org/10.1007/978-3-030-72019-3%5C_13 (visited on 03/14/2025).

Kiselyov, Oleg (2014). "The Design and Implementation of BER MetaOCaml: System Description". In: *Functional and Logic Programming*. Ed. by Michael Codish and Eijiro Sumii. Red. by David Hutchison et al. Vol. 8475. Cham: Springer International Publishing, pp. 86–102. DOI: 10.1007/978-3-319-07151-0_6. URL: http://link.springer.com/10.1007/978-3-319-07151-0_6 (visited on 02/16/2024).

— (2023). "MetaOCaml Theory and Implementation". In: *CoRR* abs/2309.08207. DOI: 10.48550/ARXIV.2309.08207. arXiv: 2309.08207. URL: https://doi.org/10.48550/arXiv.2309.08207 (visited on 03/19/2025).

Laird, James (2007). "A Fully Abstract Trace Semantics for General References". In: *Automata, Languages and Programming*. Ed. by Lars Arge et al. Vol. 4596. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 667–679. DOI: 10.1007/978-3-540-73420-8_58. URL: http://link.springer.com/10.1007/978-3-540-73420-8_58 (visited on 02/16/2024).

Milner, Robin (1977). "Fully Abstract Models of Typed \emphlambda-Calculi". In: *Theor. Comput. Sci.* 4.1, pp. 1–22. DOI: 10.1016/0304-3975(77)90053-6.

Taha, Walid and Tim Sheard (Oct. 6, 2000). "MetaML and Multi-Stage Programming with Explicit Annotations". In: *Theoretical Computer Science*. PEPM'97 248.1, pp. 211–242. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00053-0. URL: https://www.sciencedirect.com/science/article/pii/S0304397500000530 (visited on 02/16/2024).

Talcott, Carolyn (Jan. 1, 1998). "Reasoning about Programs With Effects". In: *Electronic Notes in Theoretical Computer Science*. US-Brazil Joint Workshops on the Formal Foundations of Software Systems 14, pp. 301–314. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80243-9. URL: https://www.sciencedirect.com/science/article/pii/S1571066105802439 (visited on 02/16/2024).