Mechanised Semantics of Multi-stage Programming

KA WING LI, University of Cambridge, United Kingdom
MAITE KRAMARZ, University of Toronto, Canada
NINGNING XIE, University of Toronto, Canada
JEREMY YALLOP, University of Cambridge, United Kingdom

Multi-stage programming (MSP) languages such as MetaML have subtle semantics, in which familiar properties often fail to hold and hazardous interactions with other language features such as state or polymorphism abound. The ongoing incorporation of MSP features into general purpose languages makes the need to establish confidence in their design increasingly pressing.

Taking inspiration from existing MSP systems, we present a Rocq mechanisation of a core calculus for compile-time and run-time MSP with effects, $\lambda_{\text{run}}^{\$}$, formally establishing key properties such as type and elaboration soundness and phase distinction. We hope that our mechanised semantics will be a useful basis for formal study of other designs, easing the extension of existing languages with support for MSP.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Macro languages**; **Syntax**; **Semantics**; **Functional languages**; • **Theory of computation** → **Type theory**; **Program constructs**; **Program semantics**; **Program reasoning**.

Additional Key Words and Phrases: Mechanised semantics, metatheory, multi-stage programming, elaboration, type soundness, phase distinction

1 Introduction

Language support for quotation-based multi-stage programming (MSP) is increasingly found in general-purpose programming languages such as Haskell [Sheard and Peyton Jones 2002; Xie et al. 2022], Scala [Burmako 2017; Stucki et al. 2021] and OCaml [Chiang et al. 2024; Kiselyov 2014; Xie et al. 2023a], where it is used to generate programs for a wide variety of performance-sensitive domains, from numerical computations [Carette et al. 2011] to stream processing [Kiselyov et al. 2017]. Sophisticated program generators benefit from varied programming mechanisms, such as delimited control [Oishi and Kameyama 2017], overloading and higher-rank polymorphism [Yallop 2017], mutable state [Kameyama et al. 2015], and modules [Carette and Kiselyov 2011].

Unfortunately, quotation-based MSP has complex and subtle semantics, which are typically defined indirectly by elaboration (e.g. Calcagno et al. [2003b]; Kiselyov [2015]; Xie et al. [2023a]), and where familiar properties such as the β rule do not always hold [Inoue and Taha 2016]. Worse, quotations and antiquotations (splices) introduce potentially hazardous interactions with many other language features, including mutable state [Calcagno et al. 2003a; Kiselyov et al. 2016], control

Authors' Contact Information: Ka Wing Li, University of Cambridge, , United Kingdom, Ka-Wing.Li@cl.cam.ac.uk; Maite Kramarz, University of Toronto, , Canada, maite@cs.toronto.edu; Ningning Xie, University of Toronto, , Canada, xnningxie@gmail.com; Jeremy Yallop, University of Cambridge, , United Kingdom, jeremy.yallop@cl.cam.ac.uk.

effects [Kameyama et al. 2009, 2011b], static overloading [Xie et al. 2022], polymorphism [Kiselyov 2015], and modules [Yallop and White 2015]. Modelling these interactions and ensuring that they are sound involves establishing properties such as phase distinction and generation-time binding erasure [Xie et al. 2023a] that do not typically arise in unstaged settings.

In this context, mechanising semantics can deliver substantial benefits, making it possible to establish the distinctive properties of multi-stage programming languages with high confidence, to identify and correct shortcomings in existing work, and to reduce the cost of safely extending existing designs with new features via proof engineering techniques.

This paper presents a mechanised semantics in Rocq for a family of quotation-based MSP calculi with effects, including a novel calculus $\lambda_{run}^{\$}$ that combines both run-time and compile-time MSP. The work is inspired by recently published designs for MSP in OCaml [Xie et al. 2023a] and Scala [Stucki et al. 2021] and intended as a basis for mechanising other MSP developments. The semantics of our calculus comes with proofs of key metatheoretical properties — both standard results such as type soundness, and distinctive MSP properties such as phase distinction. Where possible, the mechanisation uses standard techniques such as the locally nameless approach to variable binding [Charguéraud 2012]; where necessary, it introduces new techniques such as *stage-indexed relations* that support reasoning about references constructed at different stages.

Structure and Contributions

The rest of the paper is structured as follows: §2 presents background material on multi-stage programming (§2.1), on run-time and compile-time code generation (§2.2.1, §2.2.2) and introduces our novel calculus $\lambda_{run}^{\$}$ which supports both forms, along with its mechanised semantics, key properties, and extensions (§2.2.3, §2.3, §2.2.5).

The subsequent sections present contributions:

- §3 formally presents the static and dynamic semantics of $\lambda_{run}^{\$}$, which we believe to be the first formal calculus combining compile-time and run-time code generation.
- After a brief overview of our approach to mechanising MSP semantics (§4), §5 presents the results we have established in Rocq, including three key MSP metatheoretical properties: type soundness (§5.2), elaboration soundness (§5.3) and phase distinction (§5.4). There is very little existing work on mechanising MSP semantics, and we believe that ours is the first that goes beyond quotes and splices to support the fundamental code generation operations that are essential for practical use of MSP.

  Along the way, the exposition touches on some improvements to Xie et al.'s [2023a] work arising from mechanisation: we proved the rather subtle substitution lemma that Xie et al. assumed, and identified and fixed a bug in the statement of elaboration soundness (§5.3.1).

Finally, §6 surveys related work on MSP mechanisation and on MSP calculi that support both compile-time and run-time staging, and §7 offers directions for further work.

2   Background and Overview

This section first reviews the background of multi-stage programming, discusses complexities in its semantics, and motivates mechanisation (§2.1). It then presents an overview of the calculi with code generation (§2.2), and summarizes key mechanisation properties (§2.3).

2.1   Multi-Stage Programming

Two fundamental constructs in multi-stage programming (MSP), especially in MetaML-family languages [Taha and Sheard 2000], are the *quote* $\langle t \rangle$ and *splice* $\$t$ annotations, which allow programs

to generate and manipulate code dynamically. In a two-level calculus, quotes and splices support switching between *object-level* and *meta-level* as visualised in Fig. 1 [Calcagno et al. 2003a].

$$\frac{\Gamma \vdash_{\text{object}} t : T}{\Gamma \vdash_{\text{meta}} \langle\, t\, \rangle : \langle\, T\, \rangle} \qquad\qquad \frac{\Gamma \vdash_{\text{meta}} t : \langle\, T\, \rangle}{\Gamma \vdash_{\text{object}} \$\, t\, : T}$$

(a) Quote   (b) Splice

Fig. 1. Key constructs in MSP. The colours blue and red represent the meta and object stage respectively.

Quoting an expression $\langle t \rangle$ postpones the computation of t to a future stage by constructing a code fragment (abstract syntax tree) representing t rather than evaluating it. Splicing an expression $\$ t$ evaluates t and then integrates the resulting code fragment into the enclosing fragment, allowing the construction of a code fragment from code fragments constructed elsewhere in the program.

Formally, the use of quotes and splices naturally introduces the concept of *levels*, which typically appear as an index (i.e. an integer number) in the typing judgment, and are incremented and decremented by quotes and splices, respectively [Taha and Sheard 2000]. A key property of MSP is *the level restriction* (or *phase consistency principle* [Stucki et al. 2018]): the level where each variable x is used must match the level of the lambda-abstraction in which x is bound.

Example. We demonstrate code generation via the power function, a kind of "Hello World" example for MSP. Fig. 2a presents the pow function, which computes the nth power of a number x in a straightforward unstaged manner by repeated multiplication. The pow5 function applies pow to its argument x and the fixed exponent 5, so that calling pow5 will trigger a sequence of recursive function calls at run-time.

We now demonstrate how to eliminate the overhead of the recursive calls and branches using staging annotations in Fig. 2b. The staged pow function is identical to the unstaged pow, except for staging annotations. When executed it recursively generates a quoted expression. The corresponding pow5 returns a quotation, which takes x, and splices the result of applying pow to $\langle x \rangle$ and 5. After evaluation, pow5 generates the code in the annotation, where the function consists of five multiplications of x.

This example illustrates the power and flexibility of MSP, where code generation produces efficient code for further execution. Note that the quote and splice annotations support constructing larger quotations from smaller ones, but not extraction or execution of the constructed code fragments; §2.2 discusses additional constructs for executing generated code at run-time and at compile-time.

The subtleties of MSP. Although MSP has been adopted in various languages [Burmako 2017; Chiang et al. 2024; Kiselyov 2014; Sheard and Peyton Jones 2002; Stucki et al. 2021; Xie et al. 2022, 2023a], it has

```
1 (* unstaged *)
2 def pow = lam x. fix pow. lam n.
3   if n == 0 then 1
4   else x * pow (n-1)
5
6 def pow5 = lam x. pow x 5
```

(a) Unstaged power function

```
1 (* quotation-based staging *)
2 def pow = lam x. fix pow. lam n.
3   if n == 0 then <1>
4   else <$x * $(pow (n-1))>
5
6 def pow5 = <lam x. $(pow <x> 5)>
7 (* generates <lam x.
8     x * x * x * x * x * 1> *)
```

(b) Staged pow function

Fig. 2. Multi-stage programming

complex semantics and its interaction with many language features can introduce subtleties. In §5.3.1, we discuss how our mechanisation revealed a bug in a prior pen-and-paper proof.

We argue that the intricacy of MSP's semantics and its complex potential interactions with other language features makes mechanising MSP semantics both challenging and valuable. There is little work on mechanising its semantics in theorem provers. To our knowledge, the only existing work of this type is by Kameyama et al. [2011a,b], who have mechanised a calculus that supports MSP and delimited control but does not offer a mechanism for executing the generated code.

This paper: mechanisation of MSP. This paper presents the first fully mechanised formalization of quotation-based MSP *with* code generation; we provide details of the mechanisation in §4. We build up our proof development incrementally, mechanising six calculi in total, whose hierarchy is given in Fig. 3. Note that in the diagram, dashed boxes indicate language extensions which are interesting or common features in the MSP literature, but are not our core. In contrast, the solid arrows describe the hierarchy of features to whose analysis we devote the bulk the paper.

We start with $\lambda_{\langle\rangle}$ (§2.1), a basic quotation-based staging calculus. This calculus features a level-indexed type system and employs *level-indexed reduction* [Calcagno et al. 2003b; Taha and Sheard 2000], but lacks explicit code generation mechanisms, similar to Kameyama et al. [2011a,b]. We then mechanise the calculi extended with two different code generation techniques:

(1) λ_{run}, which adds MetaML-family style run-time code generation (§2.2.1)
(2) $\lambda^{\$}$, which adds Template Haskell/ MacoCaml style compile-time code generation (§2.2.2)

These two calculi are not new in themselves, but their corresponding semantics are mechanised for the first time, providing a valuable foundation for further development. To demonstrate this, we mechanise language extensions for each of the two calculi, described in §2.2.5 and §2.2.6. We intend for this mechanisation to make it possible to establish distinctive MSP properties with high confidence.

Fig. 3. Hierarchy of calculi

While most existing systems exclusively use either run-time or compile-time code generation, each mechanism offers distinctive advantages. This motivates a natural desire to *combine* both mechanisms within a unified system. To this end, we propose a novel calculus that, for the first time, features both run- and compile-time code generation. The design presents challenges, as we will demonstrate in §2.2.3. By fully mechanising the properties of this calculus, we establish its soundness and rigorously verify its desired properties.

2.2 Code Generation

We begin by introducing run-time code generation (§2.2.1), followed by compile-time code generation (§2.2.2). We then present a novel combination of these two code generation mechanisms (§2.2.3), and illustrate its use with an example (§2.2.4). Finally, in (§2.2.5) and (§2.2.6) we discuss two extensions of the run-time and compile-time calculi respectively.

2.2.1 ***Run-time code generation*** *(λ_{run}).* MetaML-family languages [Taha and Sheard 2000] support run-time code generation via a run construct. Fig. 4 presents a corresponding pow example.
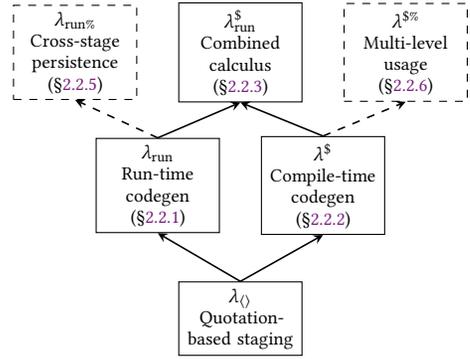
The pow function remains the same as before (Fig. 2b). On the other hand, pow5 (Fig. 4a) runs the quotation. The run construct triggers runtime code generation during evaluation, extracting a quoted program from its quotation and evaluating it. In this case, run simply takes the expanded lambda outside of the quotation.

Fig. 4b presents the generated pow5 definition after evaluation, where pow5 is now a function rather than a quotation.

```
1 (* run-time code generation *)
2 def pow = (* as before *)
3 def pow5 = run <lam x. $(pow <x> 5)>
```

(a) Run-time code generation

```
1 def pow5 = lam x.x * x * x * x * x * 1
```

(b) Generated pow5 function

Fig. 4. Run-time code generation

The run construct introduces complexities to the formalism, as it dynamically changes the level of an expression, and evaluates expressions within quotations. This creates the risk of evaluating *open code*, such as in run <x>, which would get stuck because x cannot be evaluated. Taha [1999, §4] presented an early version of MetaML where type soundness did not hold. In this work, we follow the approach of Taha [1999]: when type-checking run e, we increment the level of every local variable in the context by 1 during the type-checking of e.[1] We mechanise a calculus with run for run-time code generation, and establish type safety by proving syntactic type soundness.

2.2.2 *Compile-time code generation* ($\lambda^\$$).

On the other hand, languages like Template Haskell [Sheard and Peyton Jones 2002], Scala [Stucki et al. 2021] and MacoCaml [Xie et al. 2023a] offer compile-time code generation via *top-level splices*. This capability enables the development of libraries that can specialize themselves according to compile-time configurations.

Fig. 5 demonstrates compile-time code generation, again using the power function. The example combines Template Haskell's top-level splices with MacoCaml's notion of *macros*. As before, the staged mpow (5a) represents the computation of a quoted expression that computes the power function. However, notably, mpow is a *macro* defined using the mac keyword.

The corresponding mpow5 *splices* the result of applying mpow to its parameter and to 5. The splice, not surrounded by quotes, is called a *top-level splice*. A *source* program is compiled by evaluating all top-level splices to produce a *core* program. Notably, this evaluation occurs even within lambda bodies. In this case, the

```
1 (* compile-time code generation *)
2 mac mpow = lam x. fix mpow. lam n.
3   if n == 0 then <1>
4   else <$x * $(mpow (n-1))>
5
6 def mpow5 = lam x. $(mpow <x> 5)
```

(a) Staged power function with macros

```
1 mac mpow = (* as before *)
2 def mpow5 = lam x. x * x * x * x * x * 1
```

(b) After compilation

Fig. 5. Compile-time code generation

compilation generates the mpow5 function in Fig. 5b, which is now recursion-free and consists of five multiplications of *x*. As this example shows, compile-time code generation relies on an elaboration process that compiles a source program (Fig. 5a) to a core program (Fig. 5b), by evaluating top-level splices.

Following Xie et al. [2023a], we model macros as *compile-time bindings*. Formally, macro definitions (mac) are bindings typed at level -1 and can only be used at that level. In contrast, normal definitions (def) are typed at level 0, and can be used at any positive levels (a form of *cross-stage persistence*). This allows mpow5 to splice mpow in its definition.

---

[1]If only relative levels matter, this is equivalent to decreasing the typing level. However, this design fits nicely in the presence of global definitions, such as macros (§2.2.2), as we will see in §2.2.3.

Unlike Template Haskell and Maco-Caml, we support nested quotes and splices, where only topmost top-level splice triggers compile-time code generation. To identify these top-level splices, we use *compiler modes*, following Sheard and Peyton Jones [2002]. As we will demonstrate, identification of top-level splices also plays a crucial role in combining runtime and compile-time code generation. The ability to nest quotes and splices proves useful when programming with top-level splices (§2.2.4). For instance, a top-level splice that yields a

```
1 def back = lam f. <lam x. $(f <x>) >
2 def pow4 = run (back  (* run-time codegen *)
3    (lam x.pow x ($(mpow <2> 2))))
4                       (* compile-time codegen *)
```

(a) A program with run and a top-level splice

```
1 def back = (* as before *)
2 def pow4 = run (back (lam x. pow x (2 * 2 * 1)))
```

(b) After compilation

```
1 def pow4 = lam x. x * x * x * x * 1
```

(c) Generated pow4 function at run-time

Fig. 6. Combine run-time and compile-time code generation

code expression after code generation requires the source expression to contain nested quotes and, consequently, nested splices.

However, compile-time code generation introduces challenges, especially in the presence of compile-time references. Since compile-time computations can create references, it is crucial to ensure that locations allocated during compilation do not appear in the generated code, as the compile-time heap is no longer available at run time. Establishing this property is subtle. For instance, while this property does hold in MacoCaml, their statement of *elaboration soundness* [Xie et al. 2023a] *fails* to establish it due to a missing precondition. We present a concrete counterexample to the elaboration soundness theorem (§5.3.1), and then fix it.

2.2.3 The first formalism combining run-time and compile-time code generation $(\lambda^{\$}_{run})$. In this paper, we also present the first staging calculus that supports both run-time and compile-time code generation. The novel formalism combines run-time code generation via the run construct, with compile-time code generation through top-level splices and macros. The power examples (Fig. 4a and 5a) remain valid programs within this calculus. Moreover, Fig. 6 presents a power example using both run and a top-level splice, illustrating their combination.

The program in Fig. 6a uses the pow (Fig. 2b) function and the mpow macro (5a), with both a run construct and a top-level splice[2]. During compilation, the program evaluates the top-level splice on line 3, elaborating to the pow4 definition in Fig. 6b. At run-time, the run construct evaluates back, and extracts the result from the quotation and evaluates it, finally generating the pow4 definition in Fig. 6c.

Challenges. Unfortunately, combining the two code generation mechanisms poses significant challenges. As discussed, staging is formally managed through levels. Prior sections have outlined a set of constraints and design decisions:

(D1) Quotes and splices increment and decrement the typing level, respectively (§2.1);
(D2) Type-checking run increments the level of each local variable in the context (§2.2.1);
(D3) def definitions are typed at level 0 and accessible at any positive level; mac definitions are typed and used at level -1, enabling their erasure after compilation; top-level splices trigger compile-time code generation (§2.2.2).

---

[2]The back function is found in Taha [1999] and well studied in the partial evaluation literature; it converts a function from code to code into a single code fragment, and may be viewed as a two-level eta-expansion.

```
1 mac m1 = lam x.
2   if x == 0 then <<1+2>>
3   else <<2+3>>
4 def k1 = run $(m1 2)
5   (* run <2+3> --> 5*)
```

(a) good

```
1 mac m2 = lam x. <lam y. <$y + 1>>
2 def k2 = lam z. run ($(m2 ()) <z>)
3   (* lam z. run ((lam y. <$y + 1>) <z>)> *)
```

(b) good

```
1 mac m3 = lam x. getCompileFlag (); <1>
2 def k3 = run (m3 ())
3   (* run (m3 ()) --> error *)
```

(c) bad

```
1 mac m4 = lam x. <run (x ())>
2 def k4 = $(m4 m3)  (* as k3 *)
```

(d) bad

```
1 def k5 = $((lam x.
2   <run (x ())>) m3)  (* as k3 *)
```

(e) bad

Fig. 7. Example programs of subtle interactions between different code generation mechanisms

Now we consider a few examples to demonstrate how these design decisions interact with each other, as well the subtleties in combining the two code generation mechanisms:

First, consider the program in Fig. 7a. Here, k1 type-checks since while run increments the level of local variables in the context (D2), the level of macros remain at -1 (D3). The top-level splice in k1's definition is evaluated during compilation, generating run <2+3>. This generated program then evaluates at run-time, producing 5. Similarly, in Fig. 7b, splicing m2 () produces a lambda lam y. <$y + 1>. The complete compiled program is a lambda that takes an argument z, and runs and evaluates to z + 1. Note how these two examples use nested quotations, top-level splices, and run: they show compile-time code generation producing code for later run-time code generation, a valuable pattern for combining code generation techniques, similar to the pow4 example in Fig. 6.

Next, Fig. 7c presents a problematic program. Specifically, macros are compile-time bindings (D3), and they may use information available only during compilation, such as compilation flags. Thus, macros should not be accessible at run-time, and we cannot evaluate a macro at run-time. Fortunately, we can readily reject this program: since k3 is type-checked at 0, and run does not change the typing level and macros remain at level -1, m3 is not well-typed at level 0.

Unfortunately, relying solely on typing levels also presents difficulties. Consider the program in Fig. 7d. In this case, run in m4 does not directly take a macro, but rather a local variable x, which is well-typed since the x's level gets incremented (D2) and is thus used correctly within the quotation (D1). k4 is also well-typed and splices macros within a top-level splice. However, expanding the top-level splice in k4's definition generates run (m3 (); <1>) after compilation, leading us back to the problematic scenario in Fig. 7c! Moreover, we can also inline the definition of m4 within k4, which leads us to k5 in Fig. 7e, which similarly generates the problematic run (m3 (); <1>).

Analysing Fig. 7d and 7e, we observe that both examples use run within a top-level splice, either indirectly through splicing a macro that contains run (Fig. 7d) or directly (Fig. 7e). Therefore, to rule out these examples, we incorporate an additional design decision when combining the two code generation mechanisms:

(D4) We disallow the use of run within macro definitions and top-level splices.

This design is justified by the nature of run as a run-time construct, which should not be allowed during compile-time evaluation. Rather than enforcing this restriction through a potentially ad-hoc syntactic constraint, we adopt a more principled approach: our typing judgment tracks the

```
1  (* iprod : int list code -> int list code code -> int -> int code code *)
2  mac iprod = lam v. lam w. fix iprod. lam n.
3    if n > 0
4    then < < $(lift (nth $v $(lift n))) * nth $$w $(lift $(lift n))
5            + $$(iprod (n-1))> >
6    else < < 0 > >
7
8  (* compile-time code generation: specialise size *)
9  def iprod3 = lam v. lam w. $(iprod <v> <w> 3)
10
11 (* run-time code generation: specialise first vector *)
12 def iprod3d = run < lam w. $(iprod3 [1;2;3] < w >) >
13
14 (* provide the second vector *)
15 def res = iprod3d [4;5;6]  (* 32 *)
```

Fig. 8. Staging the inner product of two vectors using both compile-time and run-time code generation

typing level z, called the *stage*, and the evaluation level i, called the *level*. Intuitively, while macros and top-level splices are typed at level -1, their actual compile-time evaluation occurs at level 0, following the approach in Sheard and Peyton Jones [2002]; Xie et al. [2023a]. As such, the difference between the typing stage z and evaluation level i indicates whether the expression will be evaluated at compile-time ($i > z$) or run-time or a future stage ($i \leq z$). Therefore, the key design principle within the type system is to permit run only when $i \leq z$.

In §3, we formalise $\lambda_{run}^{\$}$, a staging calculus with fixpoints, quotations and splices, mutable integer references, and compile-time code generation through macros and top-level splices, and run-time code generation through run, incorporating design decisions (D1)-(D4). Our mechanisation proves key properties, including soundness of compile-time and run-time code generation, as well as phase distinction.

2.2.4 ***Example***. We consider a *three-stage* program that computes the inner product of two vectors [Taha 1999] . We apply both compile-time and run-time code generation, specializing the vector size at compile time and the first vector at run time. This program exemplifies a potentially practical application where shape/type information is often available during compilation, while concrete values may remain unknown until run time. Fig. 8 presents the program.

The macro iprod takes two vectors v and w along with their size n, and calculates their inner product. We assume lift : int -> int code that lifts an integer to an integer code. In the first stage, iprod3 knows the size of the two vectors, which enables specialization of the inner product function on that size, eliminating a looping overhead. In the second stage, iprod3d passes in the first vector at run-time, specializing iprod3 by eliminating the overhead of looking up the elements of the first vector each time. In this case, we know the vector statically, but more generally the vector may only become available after run-time evaluation. In the last stage, res applies the specialized iprod3d to the second vector, and calculates the final result 32.

2.2.5 ***Extending the run-time calculus with cross-stage persistence*** *($\lambda_{run\%}$)*. Cross-stage persistence (CSP) is a common feature in MSP, allowing variables to be used not just in the stage they are bound, but also in future stages as well [Hanada and Igarashi 2014]. As an example, the function `lam f. <lam x. f x >` requires cross-stage persistence to type, because if the entire expression lives at level n, then f is bound at level n and used at level $n + 1$. Note however that

CSP is poorly suited to a calculus with compile-time code generation because it violates phase distinction. To demonstrate the extensibility of this calculus, we implemented CSP in the run-time calculus, and proved that it satisfies the same safety properties.

2.2.6 ***Extending the compile-time calculus with multi-level usage*** $(\lambda^{\$\%})$. MacoCaml incorporates shifted module imports, which allow top-level declarations to be used at a lower level relative to the current module [Xie et al. 2023b]. It is possible to adopt a similar concept by allowing top-level definitions to be used at any level, and macros at any negative level. This extension has a few interesting consequences: First, definitions now act like "run-time macros," easing the burden of duplicating definitions as macros, similarly to the runtime libraries that can be both specialized at compile-time and linked at run-time [Stucki et al. 2021]. Second, we allow macro specialisation by splicing a macro within a macro. The iprod3 function in Fig. 8 can be defined with macros and be erased after compilation. Analogously to CSP, this extension is poorly suited to a calculus with run-time code generation because it violates phase distinction, and so we implement multi-level use as an extension to the compile-time calculus.

2.3 Mechanised Properties

We have used our mechanised semantics to prove several desirable properties of our calculi, and we summarize the key results in this section. All theorems and lemmas are proved in Rocq. Because our proof development has multiple calculi, it should be noted that the combined calculus ($\lambda_{run}^{\$}$), which subsumes the quote-only, run-time, and compile-time calculi, is the primary focus of future discussion in this paper. Except where explicitly stated otherwise, subsequent discussion of properties and theorems concerns this combined calculus, and we will note when we discuss aspects of our extension calculi.

Soundness of quotation-based staging and run-time code generation (§2.1 & §2.2.1). Our first key result is that evaluation of well-typed, quotation-based staging programs does not get stuck — that is, the calculus enjoys type soundness. We then extend the result to the calculus with run for run-time code generation. With MSP, type soundness relies on *well-stagedness*; that is, a variable is used only at the level to which it is bound.

This property is important for usability, freeing programmers from the need to examine *generated* code for typing errors, which can be difficult to fix. Instead, they can focus on on ensuring that the programs that they write are type-correct.

Soundness of compile-time code generation (§2.2.2). As described in §2.2.2, top-level splices generate code during compilation. Consequently, we have a source calculus and a core calculus, and compiling a source program evaluates top-level splices, generating a core program. We first establish *compile-time type soundness* for the core calculus. That means, if the expression is well-typed at level -1, its evaluation at evaluation level 0 does not get stuck. This ensures that the evaluation of expressions *during compilation* produces well-typed and well-scoped expressions.

We then prove *elaboration soundness*; that is, well-typed source programs generate well-typed core programs. Combined with the type soundness of the core calculus, elaboration soundness also implies type soundness of the source calculus.

In the presence of compile-time side-effects such as references, elaboration soundness requires properties beyond type preservation. In particular, since compile-time code generation may allocate references, a program may require a compile-time heap. That compile-time heap is typically discarded after compilation, and the generated program is run in a different environment. It is therefore necessary for generated programs to be well-typed *under an empty heap*. Running the generated program may also create further references, allocated in a run-time heap.
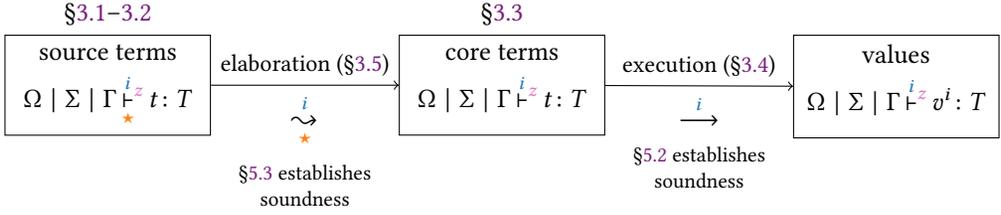
§3.1–3.2 §3.3

source terms	elaboration (§3.5)	core terms	execution (§3.4)	values
$\Omega \mid \Sigma \mid \Gamma \vdash^{i\,z} t : T$	$\overset{i}{\leadsto}$	$\Omega \mid \Sigma \mid \Gamma \vdash^{i\,z} t : T$	$\overset{i}{\longrightarrow}$	$\Omega \mid \Sigma \mid \Gamma \vdash^{i\,z} v^i : T$

§5.3 establishes
soundness

§5.2 establishes
soundness

Fig. 9. Overview of the presentation of the $\lambda_{\mathrm{run}}^{\$}$ calculus

At a high-level, reasoning about compile-time heaps requires us to reason about references created at specific stages. In other words, we need to prove that references allocated at compile-time can only be used at compile-time, and thus the generated program cannot refer to any compile-time references. As such, we can discard the compile-time heap after compilation.

MacoCaml [Xie et al. 2023a] presented a similar statement; however, MacoCaml fails to establish the proper property due to a missing precondition, which we discovered during formalisation of elaboration soundness, highlighting the importance of mechanisation. Moreover, adding the missing precondition turns out to be non-trivial.

In this paper we use a *stage-indexed relation* to establish the desired property that compile-time heaps will remain at compile-time.

Technical approach: stage-indexed relations. It is common when formalising programming language to define relations inductively on *types*, then prove properties such as strong normalisation, type safety, and program equivalence by showing that the relation holds at all base types and is preserved by language constructs.

Our work makes use of a different type of relation, indexed by stages. We prove substitution and reduction lemmas by showing that the stage-indexed predicate of interest is preserved under reduction. Stage-indexed relations allow us to prove properties specific to a certain stage — for example, that references allocated at compile-time remain at compile-time. We use them to fix a shortcoming that formalisation exposed in the elaboration soundness statement of MacoCaml [Xie et al. 2023a]. We then prove and mechanise elaboration soundness of compile-time code generation.

Phase distinction. We prove a phase distinction theorem, inspired by the phase splitting result of [Harper et al. 1989], but extended to support compile-time computation. More precisely, phase distinction guarantees that compile-time computations are not needed for run-time evaluation. As a result, we show that macros can be *erased* before running a program: that is, after compilation we can discard both compile-time heaps and compile-time bindings (i.e. macros).

Soundness of combining compile-time and run-time code generation (§2.2.3). Lastly, we formalise $\lambda_{\mathrm{run}}^{\$}$, a calculus combining compile-time and run-time code generation. As discussed in §2.2.3, deriving the correct formalism is a subtle matter due to the interaction between run, macros, and top-level splices. We fully mechanise $\lambda_{\mathrm{run}}^{\$}$, demonstrating that all the aforementioned properties continue to hold. Thus, we contribute the first formalism and mechanisation of a staging calculus combining both code generation mechanisms in a provably type-sound way.

3 The Calculus

This section presents the syntax and semantics of $\lambda_{\mathrm{run}}^{\$}$. Fig. 9 gives an overview of the section structure, which is based around two relations: *execution* (§3.4) performs run-time staging by evaluating core programs that may themselves construct and run quoted expressions, while *elaboration* (§3.5)

type	$T \in \text{TYPE}$ ::= Unit $\mid \mathbb{B} \mid \mathbb{N} \mid T_1 \to T_2 \mid \text{Ref } T \mid \langle T \rangle$	
term	$t \in \text{TERM}$ ::= unit $\mid b \mid n \mid x \mid \lambda x\!:\!T.\,t \mid t_1\,t_2 \mid \mu f.\,t \mid k \mid m$	
	$\mid \text{ref } t \mid \,!\,t \mid t_1 := t_2 \mid l \mid \langle t \rangle \mid \$\,t \mid \text{run } t$	
program	$p \in \text{PROG}$ ::= def $k = t; p \mid \text{mac } m = t; p \mid t$	

Fig. 10. Syntax of $\lambda_{\text{run}}^{\$}$

performs compile-time staging, translating source programs (§3.1–3.2) into core programs (§3.3) by evaluating code within top-level splices. Looking further ahead, §5 establishes the soundness of these two relations.

3.1 Syntax of the Source Language

Fig. 10 presents the syntax of $\lambda_{\text{run}}^{\$}$.

Types (T) include ground types *unit* (Unit), *booleans* ($\mathbb{B}$), and *numbers* ($\mathbb{N}$), *functions* ($T_1 \to T_2$), *references* (Ref T), and *code fragments* ($\langle T \rangle$).

Terms (t) include constants *unit* (unit), *booleans* (b), and *numbers* (n), as well as *variables* (x), *functions* ($\lambda x\!:\!T.\,t$), *applications* ($t_1\,t_2$), and *fixpoints* ($\mu f.\,t$). There are two additional classes of identifiers for (top-level) *definitions* (k) and *macros* (m), which are distinguished syntactically from local variables x. Function parameters are type-annotated to ensure unicity of type; we should, strictly speaking, similarly annotate fixpoint parameters, but omit the annotation in the presentation to avoid clutter.

There are three operations on references: *allocation* (ref t), *dereferencing* ($!\,t$) and *assignment* ($t_1 := t_2$). During execution, references create *locations* (l), which are included in the syntax, but cannot appear in source programs.

Finally, there are three syntactic constructs for MSP: $\langle t \rangle$ quotes an expression, $\$\,t$ splices an expression, and run t runs an expression.

A program (p) is a sequence of top-level declarations of *definitions* (def $k = t; p$) and *macros* (mac $m = t; p$) followed by an *expression* t.

3.2 Static Semantics of the Source Language

The static semantics of $\lambda_{\text{run}}^{\$}$ are inspired by the formalisation of MacoCaml [Xie et al. 2023a], but differ in a number of respects. In order to ensure the soundness of run-time code generation, our semantics of $\lambda_{\text{run}}^{\$}$ tracks both stages and levels, while MacoCaml, which supports only compile-time code generation, tracks stages only.

Moreover, we split Xie et al.'s typing context into a context Γ of variable bindings and a context Ω of top-level declarations to facilitate mechanisation. Retaining the combined context would require us to prove additional properties, e.g. that the typing context contains only top-level declarations in the type soundness proof.

Fig. 11 shows the contexts used in the static semantics of $\lambda_{\text{run}}^{\$}$. $\mathbb{A}$ is the set of possible identifier names. Contexts (Γ) map local variables to their types, stages and levels. Heap types (Σ) map locations to their types, and store types (Ω) map variables to their types, with additional labels (lab) to distinguish definitions and macros. We use two pieces of notation to succinctly denote operations on finite maps with product codomains $f : A \xrightarrow{\text{fin}} (B \times C)$:

context	$\Gamma \colon \mathbb{A} \xrightarrow{\text{fin}} \textsc{type} \times (\mathbb{Z} \times \mathbb{N})$	$::= \varnothing \mid \Gamma, x \colon (T, (z, i))$
heap type	$\Sigma \colon \mathbb{A} \xrightarrow{\text{fin}} \textsc{type}$	$::= \varnothing \mid \Sigma, l \colon T$
label	$\text{lab} \in \textsc{label}$	$::= \text{def} \mid \text{mac}$
store type	$\Omega \colon \mathbb{A} \xrightarrow{\text{fin}} \textsc{type} \times \textsc{label}$	$::= \varnothing \mid \Omega, k \colon (T, \text{def}) \mid \Omega, m \colon (T, \text{mac})$

Fig. 11. Contexts for the static semantics

- $f_{op\,c'} \triangleq \{a : (b, c\ op\ c') \mid f(a) = (b, c)\}$ where op is a binary operation on C serves as a kind of *map* function. For example, $\Gamma_{+(1,1)}$ shifts both the stages and levels of variables bound in Γ upward by 1.
- $f_{R\,c'} \triangleq \{a : (b, c) \mid f(a) = (b, c) \wedge c\ R\ c'\}$, where R is a binary relation on C that serves as a kind of *filter* function. For example, $\Omega_{\neq\text{mac}}$ removes the macros from Ω.

3.2.1 Static Semantics of Expressions. Fig. 12 presents the static semantics of source expressions. The judgment $\Omega \mid \Sigma \mid \Gamma \vdash^{i\ z}_{\star} t \colon T$ indicates that in the compilation mode $\star$, the expression t has type T at the stage z and level i under the context Ω, Σ, and Γ. The compilation mode tracks whether elaboration is below a top-level splice. We ignore the compilation mode for now and explain it in more detail when we present elaboration (§3.5).

Although the syntax supports general references, we follow Xie et al. in restricting the heap to ground types, because our focus is the distinction between the compile-time heap and run-time heaps, not the scope extrusion problems that arise from references to code. The unary relation $\Vdash T$ for well-formed heap types captures this restriction.

Definition 3.1 (Well-formed Heap Types). $\Vdash T \triangleq T = \text{Unit} \vee T = \mathbb{B} \vee T = \mathbb{N}$

While scope extrusion is an important problem in multi-stage programming, it is not our focus in this work. Omitting features that might cause scope extrusion is a common approach in the literature: many published multi-staged calculi do not support mutable state at all [e.g. Calcagno et al. 2004; Ge and Garcia 2019; Inoue and Taha 2016; Kameyama et al. 2008; Taha 1999; Xie et al. 2022]. In this style, we include a limited form of mutable state that allows us to study phase distinction with compile-time staging (§5.4); retain type soundness; and avoid committing to more elaborate techniques for avoiding scope extrusion, such as a static safety check like that of Calcagno et al. [2003a], or a dynamic check like that of Lee et al. [2026]. We leave the study of higher-order mutable state (including references to code) for future work.

Returning to the rules in Fig. 12, constants of ground types such as $\mathbb{N}$ are well-typed at any mode, stage and level (T-NAT). The rules for references, such as T-REF, ensure that only references of ground type can appear in programs. T-VAR states that free variables are well-typed if they appear in context Γ with matching type T, stage z and level i — that is, this calculus is *monolithic*, and does not support arbitrary cross-stage persistence[3].

Similarly, T-DEF and T-MAC state that uses of definitions and macros are well-typed if they are associated in context Ω with matching type and label. Uses of definitions are allowed at any non-negative stage, allowing cross-stage use of top-level definitions, while macros are allowed only at stage -1. The rule for lambda abstractions (T-ABS) is essentially the same as the corresponding rule in simply typed lambda calculus (STLC), except that the entry for the bound variable x in Γ records the stage and level information alongside the type T_1; the rule for fixpoints (T-FIX) is similar.

---

[3]In our CSP-supporting extension, the premise is relaxed to require a variable at level j where $i \geq j$. No other rules differ.

$$\boxed{\text{compiler mode} \quad \star ::= C \mid Q \mid S}$$ $$\boxed{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} t : T}$$

$$\frac{}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} n : \mathbb{N}} \ \text{T-NAT} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} t : T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} \text{ref } t : \text{Ref } T} \ \text{T-REF}$$

$$\frac{\Gamma(x) = (T, (z, i))}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} x : T} \ \text{T-VAR} \qquad \frac{\Omega(k) = (T, \text{def}) \quad 0 \leqslant z}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} k : T} \ \text{T-DEF} \qquad \frac{\Omega(m) = (T, \text{mac})}{\Omega \mid \Sigma \mid \Gamma \vdash^{i-1}_{\star} m : T} \ \text{T-MAC}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma, x : (T_1, (z, i)) \vdash^i_{\star z} t : T_2}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} \lambda x : T_1 . t : T_1 \to T_2} \ \text{T-ABS} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} t_1 : T_1 \to T_2 \qquad \Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} t_2 : T_1}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} t_1 \, t_2 : T_2} \ \text{T-APP}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma, f : (T, (z, i)) \vdash^i_{\star z} t : T}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} \mu f . t : T} \ \text{T-FIX}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i+1}_{Q \, z+1} t : T}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} \langle t \rangle : \langle T \rangle} \ \text{T-QUO} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^i_{S \, z-1} t : \langle T \rangle}{\Omega \mid \Sigma \mid \Gamma \vdash^{i+1}_{Q \vee S \, z} \$ t : T} \ \text{T-SPL}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma_{+(1,1)} \vdash^i_{\star z} t : \langle T \rangle \ \ i \leqslant z}{\Omega \mid \Sigma \mid \Gamma \vdash^i_{\star z} \text{run } t : T} \ \text{T-RUN} \qquad \frac{\Omega \mid \Sigma \mid \Gamma_{+(0,1)} \vdash^0_{S \, z-1} t : \langle T \rangle}{\Omega \mid \Sigma \mid \Gamma \vdash^0_{C \, z} \$ t : T} \ \text{T-CODEGEN}$$

Fig. 12. Static semantics of source expressions

$$\boxed{\Omega \mid \Sigma \vdash_C p : T}$$

$$\frac{\begin{array}{c} k \notin \text{dom } \Omega \\ \Omega \mid \Sigma \mid \varnothing \vdash^0_{C \, 0} t : T_1 \\ \Omega, k : (T_1, \text{def}) \mid \Sigma \vdash_C p : T_2 \end{array}}{\Omega \mid \Sigma \vdash_C \text{def } k = t; p : T_2} \ \text{TP-DEF} \qquad \frac{\begin{array}{c} m \notin \text{dom } \Omega \\ \Omega \mid \Sigma \mid \varnothing \vdash^0_{C \, -1} \lambda x : T_1 . t : T_1 \to T_2 \\ \Omega, m : (T_1 \to T_2, \text{mac}) \mid \Sigma \vdash_C p : T_3 \end{array}}{\Omega \mid \Sigma \vdash_C \text{mac } m = \lambda x : T_1 . t; p : T_3} \ \text{TP-MAC}$$

$$\frac{\Omega \mid \Sigma \mid \varnothing \vdash^0_{C \, 0} t : T}{\Omega \mid \Sigma \vdash_C t : T} \ \text{TP-EXPR}$$

Fig. 13. Static semantics of source programs

expression	$t^0 \in \text{EXPR}^0 \subset \text{TERM}$	$::= \text{unit} \mid b \mid n \mid x \mid \lambda x\!:\!T.\,t^0 \mid t_1^0\,t_2^0 \mid \mu f.\,t^0 \mid k \mid m$
		$\mid \text{ref } t^0 \mid\, !\,t^0 \mid t_1^0 := t_2^0 \mid l \mid \langle t^1 \rangle \mid \text{run } t^0$
	$t^{i+1} \in \text{EXPR}^{i+1} \subset \text{TERM}$	$::= \text{unit} \mid b \mid n \mid x \mid \lambda x\!:\!T.\,t^{i+1} \mid t_1^{i+1}\,t_2^{i+1} \mid \mu f.\,t^{i+1} \mid k \mid m$
		$\mid \text{ref } t^{i+1} \mid\, !\,t^{i+1} \mid t_1^{i+1} := t_2^{i+1} \mid l \mid \langle t^{i+2} \rangle \mid \$\,t^i \mid \text{run } t^{i+1}$
value	$v^0 \in \text{VALUE}^0 \subset \text{TERM}$	$::= \text{unit} \mid b \mid n \mid \lambda x\!:\!T.\,t^0 \mid l \mid \langle v^1 \rangle$
	$v^1 \in \text{VALUE}^1 \subset \text{TERM}$	$::= \text{unit} \mid b \mid n \mid x \mid \lambda x\!:\!T.\,v^1 \mid v_1^1\,v_2^1 \mid \mu f.\,v^1 \mid k \mid m$
		$\mid \text{ref } v^1 \mid\, !\,v^1 \mid v_1^1 := v_2^1 \mid l \mid \langle v^2 \rangle \mid \text{run } v^1$
	$v^{i+2} \in \text{VALUE}^{i+2} \subset \text{TERM}$	$::= \text{unit} \mid b \mid n \mid x \mid \lambda x\!:\!T.\,v^{i+2} \mid v_1^{i+2}\,v_2^{i+2} \mid \mu f.\,v^{i+2} \mid k \mid m$
		$\mid \text{ref } v^{i+2} \mid\, !\,v^{i+2} \mid v_1^{i+2} := v_2^{i+2} \mid l \mid \langle v^{i+3} \rangle \mid \$\,v^{i+1} \mid \text{run } v^{i+2}$
program value $p_v \in \text{VALUE}_{\text{PROG}}$		$::= \text{def } k = v^0;\, p_v \mid \text{mac } m = v^0;\, p_v \mid v^0$

Fig. 14. Level-annotated expressions and values

These are the only two rules that introduce stage and level bindings into the environment, and are essential for ensuring cross-stage safety. The rule for applications (T-APP) also follows the standard (STLC) rule, but additionally requires that the subexpressions are typed at the same stage and level as the whole expression.

The three rules that relate to MSP are the only ones that involve subtle changes to stages or levels. Quotes (T-QUO) increase both stage and level, and splices (T-SPL) decrease them. The rule for run-time code generation (T-RUN) follows the approach of Taha [1999]: the argument term t is checked in a context ($\Gamma_{+(1,1)}$) in which the stage and level of each entry is incremented. The condition $i \leqslant z$ maintains elaboration soundness, by ensuring that uses of the run construct are well-typed only at run-time. In contrast, for compile-time code generation (T-CODEGEN), top-level splices decrease the stage of the term but keep the same level 0, in order to preserve the levels of the expression after code generation. To maintain soundness for top-level splices in the same way as for the run construct, we use the fact that decrementing the stage or level in a typing rule is equivalent to incrementing the stage or level in the context Γ. Top-level splices therefore require shifting the levels but not the stages of variables in the context Γ.

The figure omits many standard rules that do not change the mode, stage or level; these omitted rules are presented in Appendix A.1.

3.2.2 Static Semantics of Source Programs. Fig. 13 gives the static semantics of source programs. The judgment $\Omega \mid \Sigma \vdash_C p : T$ indicates that program p has type T under store Ω and heap Σ. Typing top-level declarations starts with mode C and level 0. Definitions and macros are typed at stage 0 and -1 respectively.

3.3 Static Semantics of the Core

The core calculus, the result of elaborating source terms, is almost identical to the source calculus, except that it does not include top-level splices, which are removed by elaboration. The typing judgment for core expressions $\Omega \mid \Sigma \mid \Gamma \vdash^i_z t : T$ indicates that the term t has type T at the stage z and level i under the contexts Γ, Σ and Ω; it is almost identical to the source expression judgment, except that it does not include the compilation mode that guides elaboration. The typing rules (not shown here, but given in Appendix A.3) are almost identical to the rules for the source, but there is no rule for top-level splices and there is an extra rule for locations TC-LOC (which are well-typed at any mode, stage and level), which cannot appear in source programs.

$$\boxed{\Omega \mid \Sigma \vdash p \colon T}$$

$$
\frac{
\begin{array}{c}
k \notin \operatorname{dom} \Omega \\
\Omega \mid \Sigma \mid \varnothing \overset{0}{\vdash}{}^{0} t \colon T_1 \\
\Omega, k \colon (T_1, \mathrm{def}) \mid \Sigma \vdash p \colon T_2
\end{array}
}{
\Omega \mid \Sigma \vdash \operatorname{def} k = t; p \colon T_2
} \ \text{TP-DEF}
\qquad
\frac{
\begin{array}{c}
m \notin \operatorname{dom} \Omega \\
\Omega \mid \Sigma \mid \varnothing \overset{0}{\vdash}{}^{-1} v^0 \colon T_1 \\
\Omega, m \colon (T_1, \mathrm{mac}) \mid \Sigma \vdash p \colon T_2
\end{array}
}{
\Omega \mid \Sigma \vdash \operatorname{mac} m = v^0; p \colon T_2
} \ \text{TP-MAC}
$$

$$
\frac{
\Omega \mid \Sigma \mid \varnothing \overset{0}{\vdash}{}^{0} t \colon T
}{
\Omega \mid \Sigma \vdash t \colon T
} \ \text{TP-EXPR}
$$

Fig. 15. Static semantics of core programs

$$
\begin{array}{lll}
\text{heap term } \sigma \colon \mathbb{A} \overset{\text{fin}}{\longrightarrow} \text{TERM} & & \coloneqq \varnothing \mid \sigma \left[l \mapsto t \right] \\[4pt]
\text{store term } \omega \colon \mathbb{A} \overset{\text{fin}}{\longrightarrow} \text{TERM} \times \text{LABEL} & \coloneqq \varnothing \mid \omega \left[k \mapsto (t, \mathrm{def}) \right] \mid \omega \left[m \mapsto (t, \mathrm{mac}) \right]
\end{array}
$$

Fig. 16. Heap terms and store terms

Fig. 14 shows the syntax of *level-annotated expressions* ($t \in \text{EXPR}^i$) and *values* ($t \in \text{VALUE}^i$) where $i \in \mathbb{N}_0$. A splice $\$\,t$ is an expression only at positive levels; we can thus use level-annotated expressions to indicate the absence of top-level splices. The property $t \in \text{VALUE}^{i+1} \iff t \in \text{EXPR}^i$ gives the key insight that a value at a higher level is an expression at a lower level. A program value p_v is a program where all top-level declarations are values at level 0.

Fig. 15 presents the static semantics of core programs. The typing judgment for core programs $\Omega \mid \Sigma \vdash p \colon T$ indicates that a program p has type T under the contexts Ω and Σ. Definitions and macros are of level 0, indicating the absence of top-level splices. Both definitions and macros are put into the store Ω during typing; however, definitions are expressions typed at stage 0 (TP-DEF), while macros are values typed at stage -1 (TP-MAC).

3.4 Dynamic Semantics of the Core

The dynamic semantics of $\lambda_{\mathrm{run}}^{\$}$ describe both the compile-time reduction of expressions within top-level splices and the run-time execution of programs.

We use *relational small-step* with *call-by-value* evaluation in core $\lambda_{\mathrm{run}}^{\$}$. Small-step semantics make evaluation order explicit, which is helpful for reasoning about the subtle evaluation of terms in a multi-stage language, and make it easier to reason about divergence (introduced by fixpoints) and about non-determinism (introduced by the rule D-REF, which picks *some* location l, not a deterministically-chosen l [Charguéraud et al. 2021]).

Fig. 16 shows the environments used in the dynamic semantics. Heap terms (σ) map locations to terms, following Scott and Strachey [1971]. Similarly, store terms (ω) map variables to terms, with additional labels to distinguish definitions and macros.

The relation $\Vdash t$ for well-formed heap terms restricts the heap to store only ground terms, analogous to the $\Vdash T$ relation for well-formed heap types.

Definition 3.2 (Well-formed Heap Terms). $\Vdash t \triangleq t = \mathrm{unit} \lor t = b \lor t = n$

The unary relations $\Vdash \sigma$ for value heap and $\Vdash \omega$ for value store capture the additional restrictions that the heap σ and store ω contain only values at level 0.

Definition 3.3 (Value Heap and Value Store). $\Vdash \sigma \triangleq \forall l, \sigma(l) = v^0$ and $\Vdash \omega \triangleq \forall a, \omega(a) = (v^0, \mathrm{lab})$

$$\boxed{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}$$

$$\frac{\Vdash \omega \quad \Vdash \sigma}{\omega \mid (\sigma, (\lambda x\!:\!T.\,t^0)\, v^0) \xrightarrow{0} (\sigma, [x \mapsto v]\, t)} \text{ D-BETA} \qquad \frac{\Vdash \omega \quad \Vdash \sigma \quad l \notin \mathrm{dom}\,\sigma \quad \Vdash t}{\omega \mid (\sigma, \mathrm{ref}\ t) \xrightarrow{0} (\sigma\,[l \mapsto t], l)} \text{ D-REF}$$

$$\frac{\Vdash \omega \quad \Vdash \sigma \quad \sigma(l) = t}{\omega \mid (\sigma, !\,l) \xrightarrow{0} (\sigma, t)} \text{ D-GET} \qquad \frac{\Vdash \omega \quad \Vdash \sigma \quad \Vdash t}{\omega \mid (\sigma, l := t) \xrightarrow{0} (\sigma\,[l \mapsto t], \mathrm{unit})} \text{ D-SET}$$

$$\frac{\Vdash \omega \quad \Vdash \sigma}{\omega \mid (\sigma, \$ \langle v^1 \rangle) \xrightarrow{1} (\sigma, v^1)} \text{ D-SPLICE} \qquad \frac{\Vdash \omega \quad \Vdash \sigma}{\omega \mid (\sigma, \mathrm{run}\ \langle v^1 \rangle) \xrightarrow{0} (\sigma, v^1)} \text{ D-RUN}$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i+1} (\sigma', t')}{\omega \mid (\sigma, \lambda x\!:\!T.\,t) \xrightarrow{i+1} (\sigma', \lambda x\!:\!T.\,t')} \text{ D-ABS}_1$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i+1} (\sigma', t')}{\omega \mid (\sigma, \langle t \rangle) \xrightarrow{i} (\sigma', \langle t' \rangle)} \text{ D-QUO}_1 \qquad \frac{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}{\omega \mid (\sigma, \$ t) \xrightarrow{i+1} (\sigma', \$ t')} \text{ D-SPL}_1$$

Fig. 17. Dynamic semantics of core expressions

3.4.1 Dynamic Semantics of Expressions. Fig. 17 A rule $\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')$ means: at level i and store ω, t with heap σ evaluates to t' with heap σ'. The rules fall into two categories:

Reduction rules perform real computation steps. $\lambda_{\mathrm{run}}^{\$}$ includes the reduction rules of STLC at level 0, with value heaps and value stores. For example in D-BETA, the two subexpressions are values at level 0, and a substitution is performed. Rule D-REF creates and modifies a mutable reference and requires that the referenced term is a value of ground type, while the D-GET rule dereferences a location l by resolving it in the heap σ. There are two MSP-specific cancelation rules: D-SPLICE, which operates at level 1, cancels a quote inside a splice if it is a value at level 1, and D-RUN, which operates at level 0, cancels a quote inside a run construct if it is a value at level 1.

Structural rules evaluate subexpressions. $\lambda_{\mathrm{run}}^{\$}$ includes the structural rules of STLC at level 0. In addition, all language constructs with subexpressions have structural rules at positive levels: a quoted term may have splices that need to be evaluated within any sub-expression. For instance, D-ABS$_1$ evaluates under lambda at levels greater than 0. In general, structural rules do not change the level when evaluating subexpressions, but there are two exceptions: D-QUO$_1$ increases the level, and D-SPL$_1$ decreases the level. The figure omits several less interesting structural rules, as well as those which operate similarly to other listed rules. Appendix A.5 gives the complete set of rules.

3.4.2 Dynamic Semantics of Programs. We lift the dynamic semantics to programs, given by the judgment $\omega \mid (\sigma, p) \longrightarrow (\sigma', p')$. This should be read as "under store ω and heap σ, program p is reduced to p' with new heap σ'." Top-level declarations are evaluated sequentially at level 0. Definitions are reduced to values and put into the store ω. In contrast, macros are not evaluated, for two reasons. First, the core typing rule for macros TP-MAC requires that macros are already

$$\boxed{\text{compiler mode} \quad \star ::= C \mid Q \mid S} \qquad \qquad \boxed{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}$$

$$\frac{\Vdash \omega \quad \Vdash \sigma}{\omega \mid (\sigma, x) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, x)} \text{ E-VAR} \qquad \frac{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \lambda x : T. t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \lambda x : T. t')} \text{ E-ABS}$$

$$\frac{\omega \mid (\sigma, t) \overset{i+1}{\underset{Q}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \langle t \rangle) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \langle t' \rangle)} \text{ E-QUO} \qquad \frac{\omega \mid (\sigma, t) \overset{i}{\underset{S}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \$ t) \overset{i+1}{\underset{Q \lor S}{\rightsquigarrow}} (\sigma', \$ t')} \text{ E-SPL}$$

$$\frac{\omega \mid (\sigma, t) \overset{0}{\underset{S}{\rightsquigarrow}} (\sigma', t') \quad \omega \mid (\sigma', t') \overset{0}{\longrightarrow}{}^{*} (\sigma'', \langle v^1 \rangle)}{\omega \mid (\sigma, \$ t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma'', v^1)} \text{ E-CODEGEN}$$

Fig. 18. Elaboration of source expressions

$$\boxed{\omega \mid (\sigma, p) \rightsquigarrow (\sigma', p')}$$

$$\frac{\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, t) \rightsquigarrow (\sigma', t')} \text{ EP-EXPR} \qquad \frac{\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma', t') \quad \omega \mid (\sigma', p) \rightsquigarrow (\sigma'', p')}{\omega \mid (\sigma, \text{def } k = t; p) \rightsquigarrow (\sigma'', \text{def } k = t'; p')} \text{ EP-DEF}$$

$$\frac{\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma', t') \quad \omega, m : (t', \text{mac}) \mid (\sigma', p) \rightsquigarrow (\sigma'', p')}{\omega \mid (\sigma, \text{mac } m = t; p) \rightsquigarrow (\sigma'', \text{mac } m = t'; p')} \text{ EP-MAC}$$

Fig. 19. Elaboration of source programs

values. Second, core programs are executed in a run-time phase which does not need macros. The independence of run-time evaluation from macros is formally established in §5.4.

3.5 Elaboration from Source to Core

Elaboration is the compile-time translation of source programs with top-level splices to core programs that do not contain top-level splices.

It is inspired by the formalism of MacoCaml [Xie et al. 2023a]. However, we depart from the MacoCaml presentation by separating the type-directed elaboration of the source language into static and dynamic semantics. This separation makes it easier to mechanise proofs as we need to reason about the heap, and more closely reflects the MacoCaml implementation, which first type-checks the source code of a program and then compiles the program to an executable.

Fig. 18 presents the elaboration of source expressions into core expressions. The judgment $\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')$ indicates that in the compilation mode $\star$, the source expression t at level i under heap σ is elaborated to a core expression t' with a new heap σ'. Elaboration operates homomorphically on the source, except on quotes and splices. The figure consequently includes only a few representative or notable rules, omitting the rules that straightforwardly elaborate subexpressions, which are given in Appendix A.7.

E-VAR says that free variables elaborate to themselves under a well-formed store and heap. E-ABS states that elaboration (unlike evaluation) goes under lambdas. The compilation mode tracks

ElabSoundness (516)	ElabSoundness (522)	ElabSoundness (523)
PhaseDistinction (135)	PhaseDistinction (137)	PhaseDistinction (135)
Source (220)	Source (220)	Source (227)
TypeSoundness (341)	TypeSoundness (343)	TypeSoundness (349)

TypeSoundness (275)	TypeSoundness (284)	TypeSoundness (286)			
Infrastructure (715)	Infrastructure (756)	Infrastructure (839)	Infrastructure (941)	Infrastructure (1069)	Infrastructure (955)
Core (306)	Core (317)	Core (318)	Core (347)	Core (350)	Core (359)
Base (444)	Base (458)	Base (458)	Base (474)	Base (474)	Base (488)

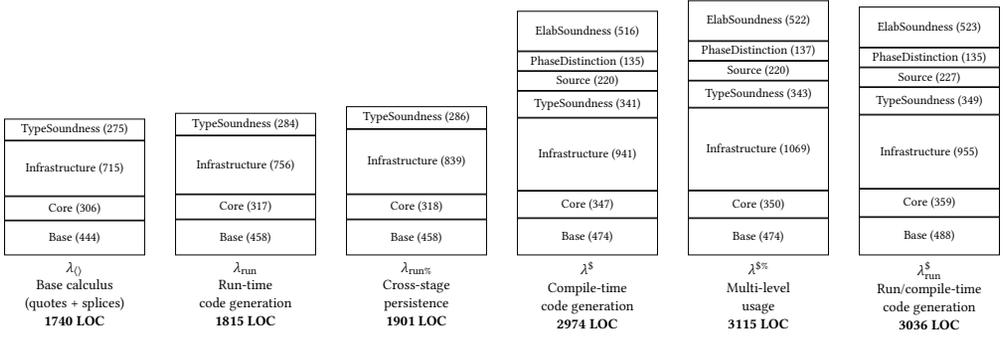| $\lambda_{\langle\rangle}$	λ_{run}	$\lambda_{\text{run\%}}$	$\lambda^{\$}$	$\lambda^{\$\%}$	$\lambda_{\text{run}}^{\$}$
Base calculus (quotes + splices) **1740 LOC**	Run-time code generation **1815 LOC**	Cross-stage persistence **1901 LOC**	Compile-time code generation **2974 LOC**	Multi-level usage **3115 LOC**	Run/compile-time code generation **3036 LOC**

Fig. 20. Lines of code in the mechanisations

top-level splices, crucial because they trigger code generation. Elaboration of expressions starts in mode C, and switches between Q and S when encountering quotes and splices. Elaborating inside the body of a quote switches compilation mode to Q, regardless of the current mode, and increments the level (E-QUO). Elaborating inside the body of a splice not at top-level, i.e. in mode Q or S, switches compilation mode to S and decrements the level (E-SPL). A splice $\$t$ that appears in mode C and at level 0 is regarded as a top-level splice and triggers code generation. The source term t is elaborated to a core term t', which is reduced to a quoted value $\langle v^1 \rangle$ using the dynamic semantics of the core, and v^1 finally extracted as a core language expression (E-CODEGEN).

Fig. 19 presents the elaboration of source programs into core programs. Elaboration is written as the judgment $\omega \mid (\sigma, p) \rightsquigarrow (\sigma', p')$, indicating that under store ω and heap σ, a source program p is elaborated to a core program p' with a new heap σ'. The top-level declarations are elaborated sequentially, in mode C at level 0. Only macros are added to the store ω because elaboration takes place in the compile-time phase, and so cannot make use of top-level definitions.

4 Mechanisation

We have formalised all the definitions of the syntax and semantics of $\lambda_{\text{run}}^{\$}$ in §3 and all the theorems, lemmas, and corollaries in §5 with Rocq version 9.0.0. Our Rocq development $\lambda_{\text{run}}^{\$}$ is fully *constructive* and *axiom-free*, both in the semantics and the proofs of the results.

4.1 Proof Development

Our mechanisation makes use of several existing tools and libraries. **Ott** [Sewell et al. 2010] is used to efficiently work on the semantics of $\lambda_{\text{run}}^{\$}$. **LNgen** [Aydemir and Weirich 2010] generates Rocq code for locally nameless representations from Ott. **LibTactics.v** [Pierce et al. 2024], contains useful tactics to facilitate proof automation. **std++** [std++ team 2024] provides data structures (e.g. finite maps) and useful lemmas and tactics.

Fig. 20 shows lines of code used per calculus. The base calculus supports multi-stage programming with quotes, splices and first-order state, and the other calculi add support for run-time staging via run and for compile-time staging via macros and top-level splices. As the figure shows, compile-time code generation is a more substantial addition to the base calculus than run-time code generation, since it introduces an elaboration step along with theorems about that elaboration.

In each case, Base.v defines the syntax of the calculus, along with various predicates on the heap and store. The static and dynamic semantics of the core and source languages are respectively defined in Core.v and (for the calculi supporting compile-time code generation) in Source.v. Infrastructure.v includes tactics and lemmas related to the locally nameless representation, as well as the

properties of the calculus. As the filenames suggest, TypeSoundness.v, ElaborationSoundness.v, and PhaseDistinction.v prove the key theorems of $\lambda_{\text{run}}^{\$}$.

4.2 Differences Between the Mechanisation and the Calculus

Representation for variable bindings. The presentation of the calculus in §3 uses the familiar notation of named variables and substitutions. However, dealing with names is inconvenient in mechanised semantics, so our mechanisation represents variable bindings using the *locally nameless* representation (LN) [Charguéraud 2012], which combines de Bruijn indices for bound variables with a nominal approach to free variables. With LN, a lambda abstraction is written as $\lambda.\,t$. Two fundamental operations switch between the de Bruijn representation and the nominal approach:

- *(opening)* $\{i \to x\}\,t$ means *open index i with atom x in t*, with shorthand $t^x \triangleq \{0 \to x\}\,t$.
- *(closing)* $\{i \leftarrow x\}\,t$ means *close atom x with index i in t*, with shorthand $^{\backslash x}t \triangleq \{0 \leftarrow x\}\,t$.

LN combines the advantages of nameless and nominal approaches: it supports simple definitions of capture-avoiding substitution and β-reduction, and avoids the need to deal with α-equivalence.

Co-finite quantification of bindings. To further ease mechanisation, we use the *co-finite quantification* rule of bound variables [Aydemir et al. 2008; Charguéraud 2012]. To understand the representation, first consider the following two typing rules:

$$
\text{WEAK} \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{STRONG}
$$

$$
\frac{x \notin \mathrm{fv}(t) \cup \mathrm{dom}(\Gamma) \quad\quad \Gamma, x : T_1 \vdash_w t^x : T_2}{\Gamma \vdash_w \lambda.\,t : T_1 \to T_2} \quad\quad \frac{\forall x \notin \mathrm{fv}(t) \cup \mathrm{dom}(\Gamma) \implies \Gamma, x : T_1 \vdash_s t^x : T_2}{\Gamma \vdash_s \lambda.\,t : T_1 \to T_2}
$$

The existential rule has a strong elimination form but a weak introduction form. Specifically, its induction hypothesis holds only for a particular x introduced by the derivation, which is often not fresh enough, necessitating need non-trivial renamings. Conversely, the universal rule has a strong introduction form but a weak elimination form, which requires its premise to hold for every sufficiently fresh name.

Aydemir et al. [2008] advocate a co-finitely quantified rule of LN:

$$
\frac{\forall x \notin A \implies \Gamma, x : T_1 \vdash_c t^x : T_2}{\Gamma \vdash_c \lambda.\,t : T_1 \to T_2} \text{ COFINITE}
$$

The rule provides a compromise between the existential rule and the universal rule: its induction hypothesis holds for infinitely many xs, and often one of them is fresh enough; moreover, we can carefully choose A to exclude problematic atoms during introduction.

It is common in MSP to manipulate open terms, such as evaluation under lambda at higher stages. By employing LN with co-finite quantification, we obtain strong induction principles for free.

A hybrid of substitutional and environmental model. Our hybrid of *substitutional* and *environmental* models for the operational semantics is inspired by the internal design of the Lean4 theorem prover [Microsoft Research 2024], which separates variable bindings from the environment for the top-level identifiers. This design decision helps with ensuring macro hygiene, as demonstrated in Fig. 21, where the top-level definition k is not captured by the lambda after compilation.

Syntactic notations as judgments. The calculus of §3 uses several simplifying syntactic conventions that avoid exposing the full bureaucratic requirements of mechanisation. First, the notation v^i in the calculus rules indicates that the expression denoted is a value at level i, but the mechanisation instead adds an extra premise $v \in \text{VALUE}^i$ to each such rule. Second, §3 defines $t \in \text{EXPR}^i$ and $v \in \text{VALUE}^i$ in Backus–Naur form, but the mechanisation defines them as level-indexed inductive

families. Finally, the LN approach requires that we add *locally-closed* predicates to the premises of binding rules in the semantics of $\lambda_{\text{run}}^{\$}$.

5 Mechanised Metatheory

This section presents the metatheoretical results that we have formally established for $\lambda_{\text{run}}^{\$}$. §5.1 describes the basic properties of the calculus, and the following three sections present the key properties: type soundness (§5.2), elaboration soundness (§5.3) and phase distinction (§5.4). The discussion of elaboration soundness for $\lambda_{\text{run}}^{\$}$ includes the identification and correction of a shortcoming that we have identified in the published statement of elaboration soundness for MacoCaml [Xie et al. 2023a] (§5.3.1) and the notion of *stage-indexed relations* (§5.3.2) that we use in our proofs.

```
1 def k = 42
2 mac m = lam _ . <k>
3 def _ = lam k . $(m 0)
```
        compile $\downarrow$
```
1 def k = 42
2 mac m = lam _ . <k>
3 def _ = lam k_1 . k
```

Fig. 21. An example of capture-avoiding macros

5.1 Basic Properties

We prove variants of two classical lemmas known as promotion and demotion [Calcagno et al. 2003a; Moggi et al. 1999], generalized to our separate treatment of typing stages and evaluation levels. These lemmas allow us to reason about cross stage usage of top-level definitions and compile-time code generation by top-level splices.

Lemma 5.1 (Promotion). *An expression that is well-typed at run-time is also well-typed at any future stage. That is, whenever $i \leqslant z \leqslant z'$, we have*

$$\Omega \mid \Sigma \mid \Gamma \overset{i}{\vdash}{}^{z} t : T \qquad implies \qquad \Omega \mid \Sigma \mid \Gamma_{+(z'-z,0)} \overset{i}{\vdash}{}^{z'} t : T$$

Lemma 5.2 (Demotion). *A value that is well-typed at compile-time is also well-typed at one lower level. That is, whenever $z \leqslant i + 1$ we have, for every $t \in \text{VALUE}^{i+1}$,*

$$\Omega \mid \Sigma \mid \Gamma_{+(0,1)} \overset{i+1}{\vdash}{}^{z} t : T \qquad implies \qquad \Omega \mid \Sigma \mid \Gamma \overset{i}{\vdash}{}^{z} t : T$$

These lemmas allow us to establish substitution properties for the static and dynamic semantics of $\lambda_{\text{run}}^{\$}$. The substitution lemmas are generally as expected and are so elided, though we note that substitution in the dynamic semantics requires an extra predicate due to the subtle interaction between stores and terms. For example, consider the reduction rule for definitions (the case for macros is similar):

$$\omega \mid (\sigma, k) \overset{0}{\longrightarrow} (\sigma, t) \qquad (\text{given } \omega(k) = t)$$

This reduction should be preserved under the substitution of an expression u for a free variable x:

$$\omega \mid (\sigma, [x \mapsto u] \, k) \overset{0}{\longrightarrow} (\sigma, [x \mapsto u] \, t) \qquad (\text{given } \omega(k) = t)$$

On the left-hand side of the reduction substitution cannot affect k, because top-level variables and lambda-bound variables are syntactically disjoint classes. However, on the right-hand side of the reduction, t could in principle contain the variable x, invalidating the reduction rule after substitution. We then extend the definition of free variables to stores, and include it as a precondition:

Definition 5.1 (Free variables of stores). $\text{fv}(\omega) \triangleq \bigcup \{\text{fv}(t) \mid \omega(a) = (t, \text{lab})\}$

Before moving to the main theorems, we give *constructive* definitions for well-formed heaps and stores, which allow us to extract *evidence* of the term associated with an identifier of known type.
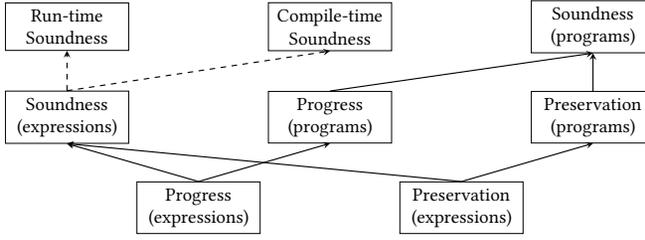
Fig. 22. Proof dependencies for type soundness theorems

Definition 5.2 (Well-formed heap).

$$\Sigma \Vdash \sigma \triangleq \begin{cases} \operatorname{dom} \sigma \equiv \operatorname{dom} \Sigma \\ \Sigma(l) = T \implies \exists t, \sigma(l) = t \land \varnothing \mid \varnothing \mid \varnothing \overset{0}{\vdash} t : T \end{cases}$$

Definition 5.3 (Well-formed store (Compile-time)).

$$\Omega \mid \Sigma \Vdash^{c} \omega \triangleq \begin{cases} \operatorname{dom} \omega \subseteq \operatorname{dom} \Omega \\ \forall a\, T, \omega(a) \neq (T, \operatorname{def}) \\ \Omega(a) = (T, \operatorname{mac}) \implies \exists t, \omega(a) = (t, \operatorname{mac}) \land \Omega \mid \Sigma \mid \varnothing \overset{0}{\vdash}{}^{-1} t : T \end{cases}$$

Definition 5.4 (Well-formed store (Run-time)).

$$\Omega \mid \Sigma \Vdash^{r} \omega \triangleq \begin{cases} \operatorname{dom} \omega \subseteq \operatorname{dom} \Omega \\ \forall a\, T, \omega(a) \neq (T, \operatorname{mac}) \\ \Omega(a) = (T, \operatorname{def}) \implies \exists t, \omega(a) = (t, \operatorname{def}) \land \left(\forall z \geqslant 0 \implies \Omega \mid \Sigma \mid \varnothing \overset{0}{\vdash}{}^{z} t : T \right) \end{cases}$$

5.2 Type Soundness

We follow the widely-used approach of establishing soundness by proving preservation and progress properties [Wright and Felleisen 1994]. The preservation lemma is reused later for proving the elaboration soundness (§5.3) and the phase distinction theorem (§5.4). The dependencies between type soundness-related theorems are given in Fig. 22, with dashed lines representing corollaries.

5.3 Elaboration Soundness

5.3.1 Counterexample. Xie et al. [2023a] present the following elaboration soundness statement:

Theorem 5.3 (MacoCaml Elaboration Soundness (Expressions)). *Elaboration of source expressions preserves typing. That is, under a well-formed heap σ and well-formed store ω, and with $z \leqslant i$,*

$$\Omega \mid \Sigma \mid \Gamma \overset{i}{\underset{\star}{\vdash}}{}^{z} t : T \quad and \quad \omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t') \qquad imply \qquad \Omega \mid \varnothing \mid \Gamma \overset{i}{\vdash}{}^{z} t' : T$$

While the MacoCaml language does have the elaboration soundness property (i.e. every well-typed source program elaborates to a well-typed core program under an empty heap), this theorem does not quite hold as stated. The problem can be demonstrated via a source configuration that elaborates to a core program l that is not well-typed with an empty heap in the core language:

$$\begin{aligned} \Omega &= \varnothing, m : \mathbb{N} \to \langle \operatorname{Ref} \mathbb{N} \rangle & \omega &= \varnothing \, [m \mapsto \lambda x : \mathbb{N}. \, \langle l \rangle] \\ \Sigma &= \varnothing, l : \mathbb{N} & \sigma &= \varnothing \, [l \mapsto 0] \\ \Gamma &= \varnothing & t &= \$ \, (m \, 42) & T &= \operatorname{Ref} \mathbb{N} \end{aligned}$$

$$\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma, l)$$

This source configuration cannot arise from a closed source program, but it might arise by linking an open source program against a non-empty heap. The failure of the theorem then arises from a missing invariant on heaps: the heap should not have values containing locations with positive stages. We now explain how to add the missing invariant and repair the theorem.

Write $\text{NoPositiveLoc}_u(t)$ to mean that a term t at stage u does not contain locations with positive stages. $\text{NoPositiveLoc}_u(t)$ traverses t, incrementing and decrementing u at quotes and splices, and fails if it encounters a location when u is positive. Appendix A.9 gives a full definition.

The invariant that the store ω contains no locations with positive stages can then be defined by checking NoPositiveLoc_0 for every t in ω:

$$\text{ok } \omega \triangleq \forall a, \omega(a) = (t, \text{lab}) \implies \text{NoPositiveLoc}_0(t)$$

Adding the requirement $\text{ok } \omega$ to the statement of elaboration soundness given by Xie et al. [2023a] addresses the problem identified above:

Theorem 5.4 (Elaboration Soundness (Source Expressions)). *Elaboration of source expressions preserves typing. That is, under a well-formed heap σ and well-formed store ω with no locations at positive stages, and with $z \leqslant i$, we have that*

$$\Omega \mid \Sigma \mid \Gamma \overset{i}{\underset{\star}{\vdash}}{}^z t : T \quad and \quad \omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t') \qquad imply \qquad \Omega \mid \varnothing \mid \Gamma \overset{i}{\vdash}{}^z t' : T$$

Theorem 5.5 (Elaboration Soundness (Source Programs)). *Elaboration of source programs preserves typing. That is, under a well-formed heap σ and well-formed store ω with no locations at positive stages, we have*

$$\Omega \mid \Sigma \underset{C}{\vdash} p : T \quad and \quad \omega \mid (\sigma, p) \rightsquigarrow (\sigma', p') \qquad imply \qquad \Omega \mid \varnothing \vdash p' : T$$

The case where the store and heap are empty follows as an easy corollary:

Corollary 5.6. *If $\varnothing \mid \varnothing \underset{C}{\vdash} p : T$ and $\varnothing \mid (\varnothing, p) \rightsquigarrow (\sigma, p')$, then $\varnothing \mid \varnothing \vdash p' : T$.*

Proving elaboration soundness in the presence of MSP involves reasoning about non-trivial properties, especially the interaction between stages and the MONOLITHIC rule. We approach the challenge using the notion of *stage-indexed relations*.

5.3.2 Stage-Indexed Relations. To prove elaboration soundness, we must ensure that compile-time computations do not propagate the heap to run time; in particular, locations must not cross to future stages. Intuitively, this property follows from the fact that locations can only be generated at level 0 of some stage z, and there is no sequence of reduction steps that can generate a code expression with a location at a level other than 0 or a stage greater than z.

To establish this formally we prove in terms of *stage-indexed relations*, inspired by the widely-used approach of logical relations, which are used to prove thorny properties of programming languages such as strong normalisation. We show that our stage-indexed predicate NoPositiveLoc is preserved under reduction and substitution, and then prove properties specific to a certain stage — for example, that references allocated at compile-time remain at compile-time. In the abstract, we show that a stage-indexed predicate $P_u(\cdot)$ is preserved under reductions, i.e.

$$\Gamma \vdash^z t : T \wedge t \overset{i}{\longrightarrow} t' \wedge P_i(t) \qquad \text{implies} \qquad P_i(t')$$

along with a substitution property, i.e.

$$\Gamma \vdash^z t : T \wedge \Gamma \Vdash^z_u \gamma \wedge P_u(t) \qquad \text{implies} \qquad P_u(\gamma(t))$$

where the notation $\Gamma \Vdash^z_u \gamma$ indicates that the substitution γ satisfies the type context Γ.

The mechanisation establishes the substitution property for NoPositiveLoc, uses it to establish the reduction property, and then uses the reduction property in the proof of elaboration soundness.

5.4 Phase Distinction

A distinctive property of MacoCaml [Xie et al. 2023a], which $\lambda_{\text{run}}^{\$}$ inherits, is phase distinction, which permits us to erase macros from both the typing context and the evaluation context before run time, without changing run-time execution.

Definition 5.5 (Macro Erasure $[[\cdot]]$).

$$
\begin{array}{l|ll}
[[\Omega]] \triangleq \Omega_{\neq\text{mac}} & [[\varnothing]] & = \varnothing \\
& [[\Omega, k : (T, \text{def})]] & = [[\Omega]], k : (T, \text{def}) \\
& [[\Omega, m : (T, \text{mac})]] & = [[\Omega]] \\
\hline
[[\omega]] \triangleq \omega_{\neq\text{mac}} & [[\varnothing]] & = \varnothing \\
& [[\omega, k : (t, \text{def})]] & = [[\omega]], k : (t, \text{def}) \\
& [[\omega, m : (t, \text{mac})]] & = [[\omega]] \\
\hline
[[p]] & [[\text{def } k = t; p]] & = \text{def } k = t; [[p]] \\
& [[\text{mac } k = t; p]] & = [[p]] \\
& [[t]] & = t
\end{array}
$$

Our mechanisation uses Rocq's instance classes to define a generic metadata filter for f_{Rc}. Erasure in the store is an instance of this filter, while erasure in a program is defined recursively on program structure. The following theorems establish that erasure preserves static and dynamic semantics.

Theorem 5.7 (Erasure (Expressions)).

Assuming (1) $\Omega \mid \Sigma \mid \Gamma \vdash_z^i t : T$ (2) $i \leqslant z$ (3) $\Omega \mid \Sigma \Vdash^r \omega$ (4) $\omega \mid (\sigma, t) \xrightarrow{i}^* (\sigma', t')$

we have (1) $[[\Omega]] \mid \Sigma \mid \Gamma \vdash_z^i t : T$ (2) $[[\omega]] \mid (\sigma, t) \xrightarrow{i}^* (\sigma', t')$

Theorem 5.8 (Erasure (Programs)).

Assuming (1) $\Omega \mid \Sigma \vdash p : T$ (2) $\Omega \mid \Sigma \Vdash^r \omega$ (3) $\omega \mid (\sigma, p) \longrightarrow (\sigma', p')$

we have (1) $[[\Omega]] \mid \Sigma \vdash [[p]] : T$ (2) $[[\omega]] \mid (\sigma, [[p]]) \longrightarrow (\sigma', [[p']])$

6 Related Work

6.1 Mechanised Semantics of Multi-Stage Programming

To our knowledge, although there is a great deal of formal pen-and-paper study of multi-stage languages, there is only one previously published mechanised semantics of such a language, described by Kameyama et al. [2011a] with additional details given in an extended abstract [Kameyama et al. 2011b]. That work studies a calculus, $\lambda^{\varnothing}$, that combines quotation with answer-type polymorphic delimited control, with a restriction that prevents moving code across binders. The Twelf mechanisation accompanying the work implements a small-step reduction over intrinsically-typed syntax, with a proof of type soundness. Since $\lambda^{\varnothing}$ does not support code execution mechanisms such as top-level splices (for compile-time code execution) or the run construct (for run-time code execution), the other properties we study here are not applicable. The extended abstract describes some (apparently unresolved) challenges with dropping the restriction on code motion across binders, arising from the representation of binding contexts and continuation frames in the semantics.

Multi-stage programming is sometimes viewed as kind of explicit variant of offline partial evaluation, in which the reduction of statically-known terms is guided by annotations rather than by an automated binding-time analysis. A form of offline partial evaluation, Type-Directed Partial Evaluation (TDPE), has been the subject of some formal study, by Ilik [2013], who shows that the TDPE algorithm normalizes well-typed programs, and by Hirota and Asai [2014], who show in 300 lines of Rocq code that TDPE for simply-typed lambda calculus additionally preserves semantics.

Taking a broader interpretation of mechanised proof, several works establish type soundness via shallow embedding of various staged calculi, inheriting soundness properties from the type system of the host language. For example, Kameyama et al. [2015] use a tagless-final embedding [Carette et al. 2009] of an effectful staged language into Haskell, Kiselyov et al. [2016] uses a similar embedding into OCaml, and Schuster and Brachthäuser [2018] uses a HOAS embedding of a two-level lambda calculus in Idris. More recently, Kovács [2024] accompanies a design for closure-free programming in a two-stage language with an embedding into Agda as a collection of axioms.

6.2 Mixed Compile-Time/Run-Time Multi-Stage Programming

We are aware of two published systems that combine compile-time and run-time MSP in a single language. Both Parreaux et al. [2017a] and Stucki et al. [2018] give informal presentations of extensions to Scala that use the same features to support both run-time MSP and compile-time macro systems. Our system is inspired by the design of Stucki et al., and we also follow them in disallowing uses of the run construct within top-level splices, but not in prohibiting effectful code within splices altogether, since our ground-type references do not introduce the scope extrusion problem that their blanket prohibition seeks to prevent. However, although the features we study are similar, the nature of our work is different: rather than presenting an informally-described language extension we study a formal calculus, firmly establishing key properties via mechanisation.

There is also unpublished work on extending compile-time MSP systems to support run-time MSP, in Typed Template Haskell [Mainland 2013] and in extensions of work by Kovács [2022].

7 Future Work

The work described here is intended as a first step towards a framework for mechanised study of multi-stage languages. The literature on such languages suggests several possible future directions.

Approaches to quotation. As another kind of extension built on this calculus, we might incorporate support for different *varieties of quotation*, such as heterogeneous [Mainland 2012] or untyped quotations [Berger et al. 2017; Sheard and Peyton Jones 2002], or quotations that can be analysed as well as generated [Jang et al. 2022; Parreaux et al. 2017b; Stucki et al. 2021].

Interactions with staging. We plan to extend the system to study features that interact with MSP, such as modules and module functors as found in MacoCaml [Chiang et al. 2024; Xie et al. 2023a], as well as higher-order state (i.e. code quotations) and delimited control, which can introduce the challenging problem of scope extrusion that has received significant study [Czarnecki et al. 2004; Kameyama et al. 2009; Kiselyov et al. 2016; Kokaji and Kameyama 2011]. Various forms of more sophisticated types also introduce subtle interactions, and we hope to study some of these by extending our system with support for overloading [Xie et al. 2022] and polymorphism [Calcagno et al. 2004; Kiselyov 2015] and perhaps with some form of dependent types [Kawata and Igarashi 2019; Kovács 2022] or Generalized Algebraic Data Types (GADTs) [4].

Extraction of a MSP code generator. Following Rao et al. [2025], one potential application of our mechanisation is the extraction of an interpreter from our proof development, which would act as an executable semantics of the multi-stage framework we have mechanised.

Acknowledgments

---

[4]The 15 December 2016 entry in the MetaOCaml Changelog records a harmful interaction between GADTs and quotation: https://github.com/metaocaml/ber-metaocaml/blob/ber-n114/ber-metaocaml-114/ChangeLog

Data-Availability Statement

We provide an artifact with the submission as supplementary material. The artifact is described in §4, and contains the constructive axiom-free Rocq mechanisation of the semantics in §3 and all the lemmas and theorems of §5. It requires Rocq 9.0.0 and the stdpp library.

References

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008-01-07) *(POPL '08)*. Association for Computing Machinery, 3–15. https://doi.org/10.1145/1328438.1328443

Brian Aydemir and Stephanie Weirich. 2010. LNgen: Tool Support for Locally Nameless Representations. 933 (2010).

Martin Berger, Laurence Tratt, and Christian Urban. 2017. Modelling Homogeneous Generative Meta-Programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:23. https://doi.org/10.4230/LIPICS.ECOOP.2017.5

Eugene Burmako. 2017. Unification of Compile-Time and Runtime Metaprogramming in Scala. https://doi.org/10.5075/epfl-thesis-7159

Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. 2003a. Closed types for a safe imperative MetaML. *J. Funct. Program.* 13, 3 (2003), 545–571. https://doi.org/10.1017/S0956796802004598

Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2004. ML-Like Inference for Classifiers. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2986)*, David A. Schmidt (Ed.). Springer, 79–93. https://doi.org/10.1007/978-3-540-24725-8_7

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003b. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2830)*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer, 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

Jacques Carette, Mustafa Elsheikh, and W. Spencer Smith. 2011. A generative geometric kernel. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, Siau-Cheng Khoo and Jeremy G. Siek (Eds.). ACM, 53–62. https://doi.org/10.1145/1929501.1929510

Jacques Carette and Oleg Kiselyov. 2011. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76, 5 (2011), 349–375. https://doi.org/10.1016/J.SCICO.2008.09.008

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2021. CPS Semantics: Smoother Nondeterminism in Operational Semantics. (June 2021). https://inria.hal.science/hal-03255472 working paper or preprint.

Arthur Charguéraud. 2012. The Locally Nameless Representation. 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Tsung-Ju Chiang, Jeremy Yallop, Leo White, and Ningning Xie. 2024. Staged Compilation with Module Functors. *Proc. ACM Program. Lang.* 8, ICFP (2024). https://doi.org/10.1145/3674649

Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. 2004. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky (Eds.). Springer, 51–72. https://doi.org/10.1007/978-3-540-25935-0_4

Rui Ge and Ronald Garcia. 2019. Refining semantics for multi-stage programming. *J. Comput. Lang.* 51 (2019), 222–240. https://doi.org/10.1016/J.JVLC.2018.10.006

Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. In *Functional and Logic Programming* (Cham) *(Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, 103–118. https://doi.org/10.1007/978-3-319-07151-0_7

Robert Harper, John C Mitchell, and Eugenio Moggi. 1989. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 341–354.

Noriko Hirota and Kenichi Asai. 2014. Formalizing a correctness property of a type-directed partial evaluator. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21,*

2014, San Diego, California, USA, Co-located with POPL '14, Nils Anders Danielsson and Bart Jacobs (Eds.). ACM, 41–46. https://doi.org/10.1145/2541568.2541572

Danko Ilik. 2013. A formalized type-directed partial evaluator for shift and reset. In *Proceedings First Workshop on Control Operators and their Semantics, COS 2013, Eindhoven, The Netherlands, June 24-25, 2013 (EPTCS, Vol. 127)*, Ugo de'Liguoro and Alexis Saurin (Eds.). 86–100. https://doi.org/10.4204/EPTCS.127.6

Jun Inoue and Walid Taha. 2016. Reasoning about Multi-Stage Programs. 26 (2016), e22. https://doi.org/10.1017/S0956796816000253

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: Metaprogramming Using Contextual Types: The Stage Where System f Can Pattern Match on Itself. 6 (2022), 39:1–39:27. Issue POPL. https://doi.org/10.1145/3498700

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the stage: from staged code to typed closures. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, Robert Glück and Oege de Moor (Eds.). ACM, 147–157. https://doi.org/10.1145/1328408.1328430

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the stage: staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, Germán Puebla and Germán Vidal (Eds.). ACM, 111–120. https://doi.org/10.1145/1480945.1480962

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011a. Shifting the stage - Staging with delimited control. *J. Funct. Program.* 21, 6 (2011), 617–662. https://doi.org/10.1017/S0956796811000256

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for impure yet hygienic code generation. *Sci. Comput. Program.* 112 (2015), 120–144. https://doi.org/10.1016/J.SCICO.2015.08.007

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011b. Mechanizing Multilevel Metatheory with Control Effects. In *Proceedings of 5th ACM SIGPLAN Workshop on Mechanizing Metatheory.*, Vol. 8.

Akira Kawata and Atsushi Igarashi. 2019. A Dependently Typed Multi-stage Calculus. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11893)*, Anthony Widjaja Lin (Ed.). Springer, 53–72. https://doi.org/10.1007/978-3-030-34175-6_4

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Oleg Kiselyov. 2015. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing. In *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015 (EPTCS, Vol. 241)*, Jeremy Yallop and Damien Doligez (Eds.). 1–22. https://doi.org/10.4204/EPTCS.241.1

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. https://doi.org/10.1145/3009837.3009880

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

Yuichiro Kokaji and Yukiyoshi Kameyama. 2011. Polymorphic Multi-stage Language with Control Effects. In *Programming Languages and Systems* (Berlin, Heidelberg), Hongseok Yang (Ed.). Springer, 105–120. https://doi.org/10.1007/978-3-642-25318-8_11

András Kovács. 2022. Staged Compilation with Two-Level Type Theory. 6 (2022), 540–569. Issue ICFP. https://doi.org/10.1145/3547641 arXiv:2209.09729 [cs]

András Kovács. 2024. Closure-Free Functional Programming in a Two-Level Type Theory. 7 (2024). Issue ICFP.

Michael Lee, Ningning Xie, Oleg Kiselyov, and Jeremy Yallop. 2026. Handling Scope Checks: A Comparative Framework for Dynamic Scope Extrusion Checks. *Proc. ACM Program. Lang.* 10, POPL, Article 39 (Jan. 2026), 30 pages. https://doi.org/10.1145/3776681

Geoffrey Mainland. 2012. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 311–322. https://doi.org/10.1145/2364527.2364572

Geoffrey Mainland. 2013. Type-Safe Runtime Code Generation with (Typed) Template Haskell. https://www.cs.drexel.edu/~gbm26/2013/05/31/type-safe-runtime-code-generation-with-typed-template-haskell/.

Microsoft Research. 2024. *Lean4/Src/Lean/Expr.Lean at 6fce8f7d · Leanprover/Lean4.* GitHub.

Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. 1999. An Idealized MetaML: Simpler, and More Expressive. In *Programming Languages and Systems* (Berlin, Heidelberg), S. Doaitse Swierstra (Ed.). Springer, 193–207. https://doi.org/10.1007/3-540-49099-X_13

Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with Control: Type-Safe Multi-Stage Programming with Control Operators. 52, 12 (2017), 29–40. https://doi.org/10.1145/3170492.3136049

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017a. Squid: type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, Heather Miller, Philipp Haller, and Ondrej Lhoták (Eds.). ACM, 56–66. https://doi.org/10.1145/3136000.3136005

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2017b. Unifying Analytic and Statically-Typed Quasiquotes. 2 (2017), 13:1–13:33. Issue POPL. https://doi.org/10.1145/3158101

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2024. *Programming Language Foundations* (version 6.7 ed.). Software Foundations, Vol. 2. Online.

Xiaojia Rao, Stefan Radziuk, Conrad Watt, and Philippa Gardner. 2025. Progressful Interpreters for Efficient WebAssembly Mechanisation. *Proc. ACM Program. Lang.* 9, POPL, Article 22 (Jan. 2025), 29 pages. https://doi.org/10.1145/3704858

Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. Typing, representing, and abstracting control: functional pearl. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development* (St. Louis, MO, USA) *(TyDe 2018)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3240719.3241788

Dana Scott and Christopher Strachey. 1971. *TOWARD A MATHEMATICAL SEMANTICS FOR COMPUTER LANGUAGES*. Technical Report PRG06. OUCL. 49 pages.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective Tool Support for the Working Semanticist. 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA, 2002-10-03) *(Haskell '02)*. Association for Computing Machinery, 1–16. https://doi.org/10.1145/581690.581691

std++ team. 2024. Iris / Stdpp. https://gitlab.mpi-sws.org/iris/stdpp.

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-Stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (New York, NY, USA, 2018-11-05) *(GPCE 2018)*. Association for Computing Machinery, 14–27. https://doi.org/10.1145/3278122.3278139

Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021*, Eli Tilevich and Coen De Roover (Eds.). ACM, 110–122. https://doi.org/10.1145/3486609.3487203

Walid Taha. 1999. Multi-Stage Programming: Its Theory and Applications.

Walid Taha and Tim Sheard. 2000. MetaML and Multi-Stage Programming with Explicit Annotations. 248, 1 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/INCO.1994.1093

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed template Haskell. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498723

Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023a. MacoCaml: Staging Composable and Compilable Macros. 7 (2023). Issue ICFP. https://doi.org/10.1145/3607851

Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023b. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.* 7, ICFP (2023), 604–648. https://doi.org/10.1145/3607851

Jeremy Yallop. 2017. Staged generic programming. *Proc. ACM Program. Lang.* 1, ICFP (2017), 29:1–29:29. https://doi.org/10.1145/3110273

Jeremy Yallop and Leo White. 2015. Modular Macros. OCaml Users and Developers Workshop.

A Full inference rules

For space reasons, the paper gives only a subset of the inference rules for the static and dynamic semantics of $\lambda_{\text{run}}^{\$}$. This appendix provides the full rules, highlighting those rules that do not appear in the main body of the paper.

A.1 Static semantics of source expressions

Excerpted in Fig. 12.

$$\boxed{\text{compiler mode} \quad \star ::= C \mid Q \mid S} \qquad\qquad \boxed{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t : T}$$

$$\frac{}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \text{unit} : \text{Unit}} \text{ T-UNIT} \qquad \frac{}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} b : \mathbb{B}} \text{ T-BOOL}$$

$$\frac{}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} n : \mathbb{N}} \text{ T-NAT} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t : T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \text{ref } t : \text{Ref } T} \text{ T-REF}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t : \text{Ref } T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \,!\, t : T} \text{ T-GET} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t_1 : \text{Ref } T \quad \Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t_2 : T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t_1 := t_2 : \text{Unit}} \text{ T-SET}$$

$$\frac{\Gamma(x) = (T, (z, i))}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} x : T} \text{ T-VAR} \qquad \frac{\Omega(k) = (T, \text{def}) \quad 0 \leqslant z}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} k : T} \text{ T-DEF} \qquad \frac{\Omega(m) = (T, \text{mac})}{\Omega \mid \Sigma \mid \Gamma \vdash^{i-1}_{\star} m : T} \text{ T-MAC}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma, x : (T_1, (z, i)) \vdash^{i}_{\star}{}^{z} t : T_2}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \lambda x : T_1 . t : T_1 \rightarrow T_2} \text{ T-ABS} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t_1 : T_1 \rightarrow T_2 \quad \Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t_2 : T_1}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} t_1 \, t_2 : T_2} \text{ T-APP}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma, f : (T, (z, i)) \vdash^{i}_{\star}{}^{z} t : T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \mu f . t : T} \text{ T-FIX}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i+1}_{Q}{}^{z+1} t : T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \langle t \rangle : \langle T \rangle} \text{ T-QUO} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{S}{}^{z-1} t : \langle T \rangle}{\Omega \mid \Sigma \mid \Gamma \vdash^{i+1}_{Q \vee S}{}^{z} \$\, t : T} \text{ T-SPL}$$

$$\dfrac{\Omega \mid \Sigma \mid \Gamma_{+(1,1)} \vdash^{i}_{\star}{}^{z} t : \langle T \rangle \quad i \leqslant z}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{\star}{}^{z} \text{run } t : T} \quad \text{T-RUN} \qquad \dfrac{\Omega \mid \Sigma \mid \Gamma_{+(0,1)} \vdash^{0}_{S}{}^{z-1} t : \langle T \rangle}{\Omega \mid \Sigma \mid \Gamma \vdash^{0}_{C}{}^{z} \$ t : T} \quad \text{T-CODEGEN}$$

A.2 Static semantics of source programs

Presented in Fig. 13; repeated here for ease of reference.

$$\boxed{\Omega \mid \Sigma \vdash_{C} p : T}$$

$$\dfrac{\begin{array}{c} k \notin \operatorname{dom} \Omega \\[4pt] \Omega \mid \Sigma \mid \varnothing \vdash^{0}_{C}{}^{0} t : T_1 \\[4pt] \Omega, k : (T_1, \text{def}) \mid \Sigma \vdash_{C} p : T_2 \end{array}}{\Omega \mid \Sigma \vdash_{C} \text{def } k = t; p : T_2} \quad \text{TP-DEF} \qquad \dfrac{\begin{array}{c} m \notin \operatorname{dom} \Omega \\[4pt] \Omega \mid \Sigma \mid \varnothing \vdash^{0}_{C}{}^{-1} \lambda x : T_1.\, t : T_1 \to T_2 \\[4pt] \Omega, m : (T_1 \to T_2, \text{mac}) \mid \Sigma \vdash_{C} p : T_3 \end{array}}{\Omega \mid \Sigma \vdash_{C} \text{mac } m = \lambda x : T_1.\, t; p : T_3} \quad \text{TP-MAC}$$

$$\dfrac{\Omega \mid \Sigma \mid \varnothing \vdash^{0}_{C}{}^{0} t : T}{\Omega \mid \Sigma \vdash_{C} t : T} \quad \text{TP-EXPR}$$

A.3 Static semantics of core expressions

These rules are discussed in §3.3, but not presented in the main body of the paper.

$$\boxed{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} t : T}$$

$$\dfrac{}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} \text{unit} : \text{Unit}} \quad \text{TC-UNIT} \qquad \dfrac{}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} b : \mathbb{B}} \quad \text{TC-BOOL}$$

$$\dfrac{}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} n : \mathbb{N}} \quad \text{TC-NAT} \qquad \dfrac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} t : T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} \text{ref } t : \text{Ref } T} \quad \text{TC-REF}$$

$$\dfrac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} t : \text{Ref } T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} \,!\, t : T} \quad \text{TC-GET} \qquad \dfrac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} t_1 : \text{Ref } T \quad \Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} t_2 : T \quad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} t_1 := t_2 : \text{Unit}} \quad \text{TC-SET}$$

$$\dfrac{\Gamma(x) = (T, (z, i))}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} x : T} \quad \text{TC-VAR} \qquad \dfrac{\Omega(k) = (T, \text{def}) \quad 0 \leqslant z}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} k : T} \quad \text{TC-DEF} \qquad \dfrac{\Omega(m) = (T, \text{mac})}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{-1} m : T} \quad \text{TC-MAC}$$

$$\dfrac{\Omega \mid \Sigma \mid \Gamma, x : (T_1, (z, i)) \vdash^{i}{}^{z} t : T_2}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}{}^{z} \lambda x : T_1.\, t : T_1 \to T_2} \quad \text{TC-ABS} \qquad \dfrac{\Omega \mid \Sigma \mid \Gamma \vdash^{z}{}^{i} t_1 : T_1 \to T_2 \quad \Omega \mid \Sigma \mid \Gamma \vdash^{z}{}^{i} t_2 : T_1}{\Omega \mid \Sigma \mid \Gamma \vdash^{z}{}^{i} t_1 t_2 : T_2} \quad \text{TC-APP}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma, f : (T, (z, i)) \vdash^{i}_{z} t : T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{z} \mu f. t : T} \quad \text{TC-FIX}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i+1}_{z+1} t : T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{z} \langle t \rangle : \langle T \rangle} \quad \text{TC-QUO} \qquad \frac{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{z-1} t : \langle T \rangle}{\Omega \mid \Sigma \mid \Gamma \vdash^{i+1}_{z} \$ t : T} \quad \text{TC-SPL}$$

$$\frac{\Omega \mid \Sigma \mid \Gamma_{+(1,1)} \vdash^{i}_{z} t : \langle T \rangle \qquad i \leqslant z}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{z} \text{run } t : T} \quad \text{TC-RUN}$$

$$\frac{\Sigma(l) = T \qquad \Vdash T}{\Omega \mid \Sigma \mid \Gamma \vdash^{i}_{z} l : \text{Ref } T} \quad \text{TC-LOC}$$

A.4 Static semantics of core programs

Presented in Fig. 15; repeated here for ease of reference.

$$\boxed{\Omega \mid \Sigma \vdash p : T}$$

$$\frac{\begin{array}{c} k \notin \text{dom } \Omega \\ \Omega \mid \Sigma \mid \varnothing \vdash^{0}_{0} t : T_1 \\ \Omega, k : (T_1, \text{def}) \mid \Sigma \vdash p : T_2 \end{array}}{\Omega \mid \Sigma \vdash \text{def } k = t; p : T_2} \quad \text{TP-DEF} \qquad \frac{\begin{array}{c} m \notin \text{dom } \Omega \\ \Omega \mid \Sigma \mid \varnothing \vdash^{0}_{-1} v^0 : T_1 \\ \Omega, m : (T_1, \text{mac}) \mid \Sigma \vdash p : T_2 \end{array}}{\Omega \mid \Sigma \vdash \text{mac } m = v^0; p : T_2} \quad \text{TP-MAC} \qquad \frac{\Omega \mid \Sigma \mid \varnothing \vdash^{0}_{0} t : T}{\Omega \mid \Sigma \vdash t : T} \quad \text{TP-EXPR}$$

A.5 Dynamic semantics of core expressions

Excerpted in Fig. 17.

$$\boxed{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}$$

$$\frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, (\lambda x : T. t^0) v^0) \xrightarrow{0} (\sigma, [x \mapsto v] t)} \quad \text{D-BETA} \qquad \frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, \mu f. t) \xrightarrow{0} (\sigma, [f \mapsto \mu f. t] t)} \quad \text{D-FIX}$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}{\omega \mid (\sigma, \text{ref } t) \xrightarrow{i} (\sigma', \text{ref } t')} \quad \text{D-REF}_1 \qquad \frac{\Vdash \omega \qquad \Vdash \sigma \qquad l \notin \text{dom } \sigma \qquad \Vdash t}{\omega \mid (\sigma, \text{ref } t) \xrightarrow{0} (\sigma [l \mapsto t], l)} \quad \text{D-REF}$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}{\omega \mid (\sigma, ! t) \xrightarrow{i} (\sigma', ! t')} \quad \text{D-GET}_1 \qquad \frac{\omega \mid (\sigma, t_1) \xrightarrow{i} (\sigma', t_1') \qquad \text{lc } t_2}{\omega \mid (\sigma, t_1 := t_2) \xrightarrow{i} (\sigma', t_1' := t_2)} \quad \text{D-SET}_1$$

$$\frac{t_1 \in \text{VALUE}^i \qquad \omega \mid (\sigma, t_2) \xrightarrow{i} (\sigma', t_2')}{\omega \mid (\sigma, t_1 \coloneqq t_2) \xrightarrow{i} (\sigma', t_1 \coloneqq t_2')} \text{ D-SET}_2$$

$$\frac{\Vdash \omega \quad \Vdash \sigma \quad \sigma(l) = t}{\omega \mid (\sigma, !\, l) \xrightarrow{0} (\sigma, t)} \text{ D-GET} \qquad \frac{\Vdash \omega \quad \Vdash \sigma \quad \Vdash t}{\omega \mid (\sigma, l \coloneqq t) \xrightarrow{0} (\sigma\,[l \mapsto t], \text{unit})} \text{ D-SET}$$

$$\frac{\Vdash \omega \quad \Vdash \sigma \quad \omega(k) = (t, \text{def})}{\omega \mid (\sigma, k) \xrightarrow{0} (\sigma, t)} \text{ D-DEF} \qquad \frac{\Vdash \omega \quad \Vdash \sigma \quad \omega(m) = (t, \text{mac})}{\omega \mid (\sigma, m) \xrightarrow{0} (\sigma, t)} \text{ D-MAC}$$

$$\frac{\Vdash \omega \quad \Vdash \sigma}{\omega \mid (\sigma, \$ \, \langle v^1 \rangle) \xrightarrow{1} (\sigma, v^1)} \text{ D-SPLICE} \qquad \frac{\Vdash \omega \quad \Vdash \sigma}{\omega \mid (\sigma, \text{run } \langle v^1 \rangle) \xrightarrow{0} (\sigma, v^1)} \text{ D-RUN}$$

$$\frac{\omega \mid (\sigma, t_1) \xrightarrow{i} (\sigma', t_1') \quad \text{lc } t_2}{\omega \mid (\sigma, t_1 \, t_2) \xrightarrow{i} (\sigma', t_1' \, t_2)} \text{ D-APP}_1 \qquad \frac{t_1 \in \text{VALUE}^i \quad \omega \mid (\sigma, t_2) \xrightarrow{i} (\sigma', t_2')}{\omega \mid (\sigma, t_1 \, t_2) \xrightarrow{i} (\sigma', t_1 \, t_2')} \text{ D-APP}_2$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i+1} (\sigma', t')}{\omega \mid (\sigma, \lambda x : T. \, t) \xrightarrow{i+1} (\sigma', \lambda x : T. \, t')} \text{ D-ABS}_1 \qquad \frac{\omega \mid (\sigma, t) \xrightarrow{i+1} (\sigma', t')}{\omega \mid (\sigma, \mu f. \, t) \xrightarrow{i+1} (\sigma', \mu f. \, t')} \text{ D-FIX}_1$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i+1} (\sigma', t')}{\omega \mid (\sigma, \langle t \rangle) \xrightarrow{i} (\sigma', \langle t' \rangle)} \text{ D-QUO}_1 \qquad \frac{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}{\omega \mid (\sigma, \$ \, t) \xrightarrow{i+1} (\sigma', \$ \, t')} \text{ D-SPL}_1$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{i} (\sigma', t')}{\omega \mid (\sigma, \text{run } t) \xrightarrow{i} (\sigma', \text{run } t')} \text{ D-RUN}_1$$

A.6 Dynamic semantics of core programs

$$\boxed{\omega \mid (\sigma, p) \longrightarrow (\sigma', p')}$$

$$\frac{\omega \mid (\sigma, t) \xrightarrow{0} (\sigma', t')}{\omega \mid (\sigma, \text{def } k = t; p) \longrightarrow (\sigma', \text{def } k = t'; p)} \text{ D-DEF}_1$$

$$\frac{\omega\,[k \mapsto v] \mid (\sigma, p) \longrightarrow (\sigma', p')}{\omega \mid (\sigma, \text{def } k = v^0; p) \longrightarrow (\sigma', \text{def } k = v^0; p')} \text{ D-DEF}_2$$

$$\frac{\omega \mid (\sigma, p) \longrightarrow (\sigma', p')}{\omega \mid (\sigma, \mathrm{mac}\ m = t; p) \longrightarrow (\sigma', \mathrm{mac}\ m = t; p')}\ \text{D-MAC} \qquad \frac{\omega \mid (\sigma, t) \xrightarrow{0} (\sigma', t')}{\omega \mid (\sigma, t) \longrightarrow (\sigma', t')}\ \text{D-EXPR}$$

A.7 Elaboration of source expressions

Excerpted in Fig. 18.

$$\boxed{\text{compiler mode}\quad \star ::= C \mid Q \mid S}\qquad\qquad\qquad\qquad\boxed{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}$$

$$\frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, \mathrm{unit}) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, \mathrm{unit})}\ \text{E-UNIT} \qquad \frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, b) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, b)}\ \text{E-BOOL}$$

$$\frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, n) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, n)}\ \text{E-NAT} \qquad \frac{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \mathrm{ref}\ t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \mathrm{ref}\ t')}\ \text{E-REF}$$

$$\frac{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, !\,t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', !\,t')}\ \text{E-GET} \qquad \frac{\omega \mid (\sigma, t_1) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t_1') \qquad \omega \mid (\sigma', t_2) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma'', t_2')}{\omega \mid (\sigma, t_1 := t_2) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma'', t_1' := t_2')}\ \text{E-SET}$$

$$\frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, k) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, k)}\ \text{E-DEF} \qquad \frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, m) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, m)}\ \text{E-MAC}$$

$$\frac{\Vdash \omega \qquad \Vdash \sigma}{\omega \mid (\sigma, x) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma, x)}\ \text{E-VAR} \qquad \frac{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \lambda x{:}T.\,t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \lambda x{:}T.\,t')}\ \text{E-ABS}$$

$$\frac{\omega \mid (\sigma, t_1) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t_1') \qquad \omega \mid (\sigma', t_2) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma'', t_2')}{\omega \mid (\sigma, t_1\ t_2) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma'', t_1'\ t_2')}\ \text{E-APP} \qquad \frac{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \mu f.\,t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \mu f.\,t')}\ \text{E-FIX}$$

$$\frac{\omega \mid (\sigma, t) \overset{i+1}{\underset{Q}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \langle t \rangle) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \langle t' \rangle)}\ \text{E-QUO} \qquad \frac{\omega \mid (\sigma, t) \overset{i}{\underset{S}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \$\,t) \overset{i+1}{\underset{Q \vee S}{\rightsquigarrow}} (\sigma', \$\,t')}\ \text{E-SPL}$$

$$\frac{\omega \mid (\sigma, t) \overset{0}{\underset{S}{\rightsquigarrow}} (\sigma', t') \qquad \omega \mid (\sigma', t') \overset{0}{\longrightarrow}{}^* (\sigma'', \langle v^1 \rangle)}{\omega \mid (\sigma, \$\, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma'', v^1)} \text{ E-CODEGEN}$$

$$\frac{\omega \mid (\sigma, t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, \text{run } t) \overset{i}{\underset{\star}{\rightsquigarrow}} (\sigma', \text{run } t')} \text{ E-RUN}$$

A.8 Elaboration of source programs

Excerpted in Fig. 19; repeated here for ease of reference.

$$\boxed{\omega \mid (\sigma, p) \rightsquigarrow (\sigma', p')}$$

$$\frac{\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma', t')}{\omega \mid (\sigma, t) \rightsquigarrow (\sigma', t')} \text{ EP-EXPR} \qquad \frac{\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma', t') \qquad \omega \mid (\sigma', p) \rightsquigarrow (\sigma'', p')}{\omega \mid (\sigma, \text{def } k = t; p) \rightsquigarrow (\sigma'', \text{def } k = t'; p')} \text{ EP-DEF}$$

$$\frac{\omega \mid (\sigma, t) \overset{0}{\underset{C}{\rightsquigarrow}} (\sigma', t') \qquad \omega, m : (t', \text{mac}) \mid (\sigma', p) \rightsquigarrow (\sigma'', p')}{\omega \mid (\sigma, \text{mac } m = t; p) \rightsquigarrow (\sigma'', \text{mac } m = t'; p')} \text{ EP-MAC}$$

A.9 Definition of NoPositiveLoc

§5.3.1 introduces the NoPositiveLoc predicate, whose full definition is below:

$$
\begin{aligned}
\text{NoPositiveLoc}_u(l) &= u \le 0 \\
\text{NoPositiveLoc}_u(\langle t \rangle) &= \text{NoPositiveLoc}_{u+1}(t) \\
\text{NoPositiveLoc}_u(\$\, t) &= \text{NoPositiveLoc}_{u-1}(t) \\
\text{NoPositiveLoc}_u(x) &= \text{true} \\
\text{NoPositiveLoc}_u(\lambda x : T.\, t) &= \text{NoPositiveLoc}_u(t) \\
\text{NoPositiveLoc}_u(t_1\, t_2) &= \text{NoPositiveLoc}_u(t) \land \text{NoPositiveLoc}_u(t) \\
\text{NoPositiveLoc}_u(\mu f.\, t) &= \text{NoPositiveLoc}_u(t) \\
\text{NoPositiveLoc}_u(\text{unit}) &= \text{true} \\
\text{NoPositiveLoc}_u(\text{ref } t) &= \text{NoPositiveLoc}_u(t) \\
\text{NoPositiveLoc}_u(!\, t) &= \text{NoPositiveLoc}_u(t) \\
\text{NoPositiveLoc}_u(t_1 \coloneqq t_2) &= \text{NoPositiveLoc}_u(t_1) \land \text{NoPositiveLoc}_u(t_2) \\
\text{NoPositiveLoc}_u(b) &= \text{true} \\
\text{NoPositiveLoc}_u(n) &= \text{true} \\
\text{NoPositiveLoc}_u(k) &= \text{true} \\
\text{NoPositiveLoc}_u(m) &= \text{true} \\
\text{NoPositiveLoc}_u(\text{run } t) &= \text{NoPositiveLoc}_u(t)
\end{aligned}
$$