Program Logics for Homogeneous Meta-Programming

Martin Berger[1] Laurence Tratt[2]

[1] University of Sussex
[2] Middlesex University

Abstract. A meta-program is a program that generates or manipulates another program; in homogeneous meta-programming, a program may generate new parts of, or manipulate, itself. Meta-programming has been used extensively since macros were introduced to Lisp, yet we have little idea how formally to reason about meta-programs. This paper provides the first program logics for homogeneous meta-programming – using a variant of MiniML$_e^\square$ by Davies and Pfenning as underlying meta-programming language. We show the applicability of our approach by reasoning about example meta-programs from the literature. We also demonstrate that our logics are relatively complete in the sense of Cook, enable the inductive derivation of characteristic formulae, and exactly capture the observational properties induced by the operational semantics.

1 Introduction

Meta-programming is the generation or manipulation of programs, or parts of programs, by other programs, i.e. in an algorithmic way. Meta-programming is commonplace, as evidenced by the following examples: compilers, compiler generators, domain specific language hosting, extraction of programs from formal specifications, and refactoring tools.

Many programming languages, going back at least as far as Lisp, have explicit meta-programming features. These can be classified in various ways such as: generative (program creation), intensional (program analysis), compile-time (happening while programs are compiled), run-time (taking place as part of program execution), heterogeneous (where the system generating or analysing the program is different from the system being generated or analysed), homogeneous (where the systems involved are the same), and lexical (working on simple strings) or syntactical (working on abstract syntax trees). Arguably the most common form of meta-programming is reflection, supported by mainstream languages such as Java, C#, and Python. Web system languages such as PHP use meta-programming to produce web pages containing JavaScript; JavaScript (in common with some other languages) does meta-programming by dynamically generating strings and then executing them using its `eval` function. In short, meta-programming is a mainstream activity.

An important type of meta-programming is generative meta-programming, specifically homogeneous meta-programming. The first language to support homogeneous meta-programming was Lisp with its S-expression based macros; Scheme's macros improve upon Lisp's by being fully hygienic, but are conceptually similar. Perhaps unfortunately, the power of Lisp-based macros was long seen to rest largely on Lisp's minimalistic syntax; it was not until MetaML [17] that a modern, syntactically rich language was shown to be capable of homogeneous meta-programming. Since then, MetaOCaml [18] (a descendant of MetaML), Template Haskell [16] (a pure compile-time meta-programming language) and Converge [19] (inspired by Template Haskell and adding features for embedding domain specific languages) have shown that a variety of modern programming languages can house homogeneous generative meta-programming, and that this allows powerful, safe programming of a type previously impractical or impossible.

Meta-Programming & Verification. The ubiquity of meta-programming demonstrates its importance; by extension, it means that the correctness of much modern software depends on the correct use of meta-programming. Surprisingly, correctness and meta-programming have received little joint attention. In particular, there seem to be no program logics for meta-programming languages, homogeneous or otherwise. We believe that the following reasons might be partly responsible.

- First, developing logics for non-meta-programming programming languages is already a hard problem, and only recently have satisfactory solutions been found for reasoning about programs with higher-order functions, state, pointers, continuations or concurrency [1, 14, 21]. Since reasoning about meta-programs contains reasoning about normal programs as a special case, program logics for meta-programming are at least as complicated as logics for normal languages.
- Second, it is often possible to side-step the question of meta-programming correctness altogether by considering only the final product of meta-programming. Compilation is an example where the meta-programming machinery is typically much more complex than the end product. Verifying only the output of a meta-programming process is inherently limited, because knowledge garnered from the input to the process cannot be used.
- Finally, static typing of meta-programming is challenging, and still not a fully solved problem. Consequently, most meta-programming languages are at least partly dynamically typed (including MetaOCaml); Template Haskell on the other hand intertwines code generation with type-checking in complicated ways. Logics for such languages are not well understood in the absence of other meta-programming features; moreover, many meta-programming languages have additional features such as capturing substitution, pattern matching of code, and splicing of types, which are largely unexplored theoretically. Heterogeneous meta-programming adds the complication of multi-language verification.

Contributions. The present paper is the first in a series investigating the use of program logics for the specification and verification of meta-programming. The aim of the series is to unify and relate all key meta-programming concepts using program logics. One of our goals is to achieve coherency between existing logics for programming languages and their meta-programming extensions (i.e. the former should be special cases of the latter). The contributions of this paper are as follows (all proofs have been omitted in the interests of brevity; they can be found in [4]):

- We provide the first program logic for a generative, homogeneous meta-programming language (PCF$_{\text{DP}}$, a variant of Davies and Pfenning's MiniML$_e^{\square}$ [6], itself an extension of PCF [8]). The logic smoothly generalises previous work on axiomatic semantics for the ML family of languages [1, 2, 9, 11, 12, 21]. The logic is for total correctness (for partial correctness see [4]). A key feature of our logic is that for PCF$_{\text{DP}}$ programs that don't do meta-programming, i.e. programs in the PCF-fragment, reasoning can be

done in the simpler logic [9, 11] for PCF. Hence reasoning about meta-programming does not impose an additional burden on reasoning about normal programs.
- We show that our logic is relatively complete in the sense of Cook [5] (Section 5).
- We demonstrate that the axiomatic semantics induced by our logic coincides precisely with the contextual semantics given by the reduction rules of PCF_{DP} (Section 5).
- We present an additional inference system for characteristic formulae which enables, for each program M, the inductive derivation of a pair A, B of formulae which describe completely M's behaviour (descriptive completeness [10], Section 5).

2 The Language

This section introduces PCF_{DP}, the homogeneous meta-programming language that is the basis of our study. PCF_{DP} is a meta-programming variant of call-by-value (CBV) PCF [8], extended with the meta-programming features of Mini-$\text{ML}_e^\Box$ [6, Section 3]. From now on we will simply speak of PCF to mean CBV PCF. Mini-$\text{ML}_e^\Box$ was the first typed homogeneous meta-programming language to provide a facility for executing generated code. Typing the execution of generated code is a difficult problem. Mini-$\text{ML}_e^\Box$ achieves type-safety with two substantial restrictions:

- Only closed generated code can be executed.
- Variables free in code cannot be λ-abstracted or be recursion variables.

Mini-$\text{ML}_e^\Box$ was one of the first meta-programming languages with a Curry-Howard correspondence, although this paper does not investigate the connection between program logic and the Curry-Howard correspondence. PCF_{DP} is essentially Mini-$\text{ML}_e^\Box$, but with a slightly different form of recursion that can be given a moderately simpler logical characterisation. PCF_{DP} is an ideal vehicle for our investigation for two reasons. First, it is designed to be a simple, yet non-trivial meta-programming language, having all key features of homogeneous generative meta-programming while removing much of the complexity of real-world languages (for example, PCF_{DP}'s operational semantics is substantially simpler than that of the MetaML family of languages [17]). Second, it is built on top of PCF, a well-understood idealised programming language with existing program logics [9, 11], which means that we can compare reasoning in the PCF-fragment with reasoning in full PCF_{DP}.

PCF_{DP} extends PCF with one new type $\langle\alpha\rangle$ as well as two new term constructors, quasi-quotes $\langle M\rangle$ and an unquote mechanism $\texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N$. *Quasi-quotes* $\langle M\rangle$ represent the code of M, and allow code fragments to be simply expressed in normal concrete syntax; quasi-quotes also provide additional benefits such as ensuring hygiene. The quasi-quotes we use in this paper are subtly different from the abstract syntax trees (ASTs) used in languages like Template Haskell and Converge. In such languages, ASTs are a distinct data type, shadowing the conventional language feature hierarchy. In this paper, if M has type α, then $\langle M\rangle$ is typed $\langle\alpha\rangle$. For example, $\langle 1+7\rangle$ is the code of the program $1+7$ of type $\langle\mathsf{Int}\rangle$. $\langle M\rangle$ is a value for all M and hence $\langle 1+7\rangle$ does not reduce to $\langle 8\rangle$. Extracting code from a quasi-quote is the purpose of the *unquote* $\texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N$. It evaluates M to code $\langle M'\rangle$, extracts M' from the quasi-quote, names it x and makes M' available in N without reducing M'. For example

$$\texttt{let}\ \langle x\rangle = (\lambda z.z)\langle 1+7\rangle\ \texttt{in}\ \langle \lambda n.x^n\rangle$$

first reduces the application to $\langle 1+7\rangle$, then extracts the code from $\langle 1+7\rangle$, names it x and makes it available unevaluated to the code $\langle \lambda n.x^n\rangle$:

$$\begin{array}{rcl} \texttt{let}\ \langle x\rangle = (\lambda z.z)\langle 1+7\rangle\ \texttt{in}\ \langle \lambda n.x^n\rangle & \to & \texttt{let}\ \langle x\rangle = \langle 1+7\rangle\ \texttt{in}\ \langle \lambda n.x^n\rangle \\ & \to & \langle \lambda n.x^n\rangle[1+7/x] \\ & = & \langle \lambda n.(1+7)^n\rangle \end{array}$$

$$\frac{(\Gamma\cup\Delta)(x)=\alpha}{\Gamma;\Delta\vdash x:\alpha} \quad \frac{\Gamma,x:\alpha;\Delta\vdash M:\beta}{\Gamma;\Delta\vdash \lambda x^{\alpha}.M:\alpha\to\beta} \quad \frac{\Gamma;\Delta\vdash M:\alpha\to\beta \quad \Gamma;\Delta\vdash N:\alpha}{\Gamma;\Delta\vdash MN:\beta}$$

$$\frac{\Gamma,f:(\alpha\to\beta);\Delta\vdash \lambda x^{\alpha}.M:\alpha\to\beta}{\Gamma;\Delta\vdash \mu f^{\alpha\to\beta}.\lambda x^{\alpha}.M:\alpha\to\beta} \quad \frac{\Gamma;\Delta\vdash M:\mathsf{Bool} \quad \Gamma;\Delta\vdash N:\alpha \quad \Gamma;\Delta\vdash N':\alpha}{\Gamma;\Delta\vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N':\alpha}$$

$$\frac{\Gamma;\Delta\vdash M:\mathsf{Int} \quad \Gamma;\Delta\vdash N:\mathsf{Int}}{\Gamma;\Delta\vdash M+N:\mathsf{Int}} \quad \frac{\varepsilon;\Delta\vdash M:\alpha}{\Gamma;\Delta\vdash \langle M\rangle:\langle\alpha\rangle} \quad \frac{\Gamma;\Delta\vdash M:\langle\alpha\rangle \quad \Gamma;\Delta,x:\alpha\vdash N:\beta}{\Gamma;\Delta\vdash \texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N:\beta}$$

Fig. 1. Key typing rules for PCF_{DP}.

Not evaluating code after extraction from a quasi-quote is fundamental to meta-programming because it enables the construction of code *other than values* under λ-abstractions. This is different from the usual reduction strategies of CBV-calculi — Section 6 discusses briefly how PCF_{DP} might nevertheless be embeddable into PCF. Unquote can also be used to run a meta-program: if M evaluates to a quasi-quote $\langle N\rangle$, the program $\texttt{let}\ \langle x\rangle = M\ \texttt{in}\ x$ evaluates M, extracts N, binds N to x and then runs x, i.e. N. In this sense, PCF_{DP}'s unquote mechanism unifies splicing and executing quasi-quoted code, where the MetaML family of languages uses different primitives for these two functions [17].

Syntax and Types. We now formalise PCF_{DP}'s syntax and semantics, assuming a set of variables, ranged over by $x,y,f,u,m,...$ (for more details see [6, 8]).

$$\begin{array}{rcl} \alpha & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \alpha\to\beta \mid \langle\alpha\rangle \\ V & ::= & \mathsf{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha\to\beta}.\lambda x^{\alpha}.M \mid \langle M\rangle \\ M & ::= & V \mid \mathsf{op}(\tilde{M}) \mid MN \mid \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \mid \texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N \end{array}$$

Here α ranges over *types*, V over *values* and M ranges over *programs*. Constants c range over the integers $0,1,-1,...$, booleans t,f, and $()$ of type Unit, op ranges over the usual first-order operators like addition, multiplication, equality, conjunction, negation, comparison, etc., with the restriction that equality is *not* defined on expressions of function type or of type $\langle\alpha\rangle$. The recursion operator is $\mu f.\lambda x.M$. The *free variables* $\mathsf{fv}(M)$ of M are defined as usual with two new clauses: $\mathsf{fv}(\langle M\rangle) \stackrel{def}{=} \mathsf{fv}(M)$ and $\mathsf{fv}(\texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N) \stackrel{def}{=} \mathsf{fv}(M)\cup(\mathsf{fv}(N)\setminus\{x\})$. We write $\lambda().M$ for $\lambda x^{\mathsf{Unit}}.M$ and $\texttt{let}\ x = M\ \texttt{in}\ N$ for $(\lambda x.N)M$, assuming that $x\notin\mathsf{fv}(M)$ in both cases. A *typing environment* $(\Gamma,\Delta,...)$ is a finite map $x_1:\alpha_1,...,x_k:\alpha_k$ from variables to types. The *domain* $\mathsf{dom}(\Gamma)$ of Γ is the set $\{x_1,...,x_n\}$, assuming that Γ is $x_1:\alpha_1,...,x_n:\alpha_n$. We also write ε for the empty environment. The *typing judgement* is written $\Gamma;\Delta\vdash M:\alpha$ where we assume that $\mathsf{dom}(\Gamma)\cap\mathsf{dom}(\Delta)=\emptyset$. We write $\vdash M:\alpha$ for $\varepsilon;\varepsilon\vdash M:\alpha$. We say a term M is *closed* if $\vdash M:\alpha$. We call Δ a *modal context* in $\Gamma;\Delta\vdash M:\alpha$. We say a variable x is *modal* in $\Gamma;\Delta\vdash M:\alpha$ if $x\in\mathsf{dom}(\Delta)$. Modal variables represent code inside other code, and code to be run. The key type-checking rules are given in Figure 1. Typing for constants and first-order operations is standard.

Noteworthy features of the typing system are that modal variables cannot be λ- or μ-abstracted, that all free variables in quasi-quotes must be modal, and that modal variables can only be generated by unquotes. [6] gives detailed explanations of this typing system and its relationship to modal logics.

The *reduction relation* $\rightarrow$ is unchanged from PCF for the PCF-fragment of PCF_{DP}, and adapted to PCF_{DP} as follows. First we define define *reduction contexts*, by extending those for PCF as follows.

$$\mathcal{E}[\cdot] \ ::= \ \ldots \mid \mathtt{let}\ \langle x \rangle = \mathcal{E}[\cdot]\ \mathtt{in}\ M$$

Now $\rightarrow$ is defined as usual on *closed* terms with one new rule.

$$\mathtt{let}\ \langle x \rangle = \langle M \rangle\ \mathtt{in}\ N \rightarrow N[M/x]$$

We write $\twoheadrightarrow$ for $\rightarrow^*$. $M \Downarrow V$ means that $M \twoheadrightarrow V$ for some value V. We write $M \Downarrow$ if $M \Downarrow V$ for some appropriate V.

By $\sqsubseteq_{\Gamma;\Delta;\alpha}$ (usually abbreviated to just $\sqsubseteq$) we denote the usual *typed contextual pre-conguence*: if $\Gamma;\Delta \vdash M_i : \alpha$ for $i = 1,2$ then: $M_1 \sqsubseteq_{\Gamma;\Delta;\alpha} M_2$ iff for all closing context $C[\cdot]$ such that $\vdash C[M_i] : \mathsf{Unit}$ $(i = 1,2)$ we have $C[M_1] \Downarrow$ implies $C[M_2] \Downarrow$. We write $\simeq$ for $\sqsubseteq \cap \sqsubseteq^{-1}$ and call $\simeq$ *contextual congruence*. Other forms of congruence are possible. Our choice means that code can only be observed contextually by running it. Hence for example $\langle M \rangle$ and $\langle \lambda x.Mx \rangle$ are contextually indistinguishable if $x \notin \mathsf{fv}(M)$, as are $\langle 1+2 \rangle$ and $\langle 3 \rangle$. This coheres well with the notion of equality in PCF, and facilitates a smooth integration of the logics for PCF_{DP} with the logics for PCF. Some meta-programming languages are more discriminating, allowing, e.g. printing of code, which can distinguish α-equivalent programs. It is unclear how to design logics for such languages.

Examples. We now present classic generative meta-programs in PCF_{DP}. We reason about some of these programs in later sections.

The first example is from [6] and shows how generative meta-programming can *stage* computation, which can be used for powerful domain-specific optimisations. As an example, consider staging an exponential function $\lambda na.a^n$. It is generally more efficient to run $\lambda a.a \times a \times a$ than $(\lambda na.a^n)\ 3$, because the recursion required in computing a^n for arbitrary n can be avoided. Thus if a program contains many applications $(\lambda na.a^n)\ 3$, it makes sense to specialise such applications to $\lambda a.a \times a \times a$. Meta-programming can be used to generate such specialised code as the following example shows.

$$\mathsf{power} \stackrel{def}{=} \mu p.\lambda n.\mathtt{if}\ n \le 0\ \mathtt{then}\ \langle \lambda x.1 \rangle\ \mathtt{else}\ \mathtt{let}\ \langle q \rangle = p(n-1)\ \mathtt{in}\ \langle \lambda x.x \times (q\,x) \rangle$$

This function has type $\vdash \mathsf{power} : \mathsf{Int} \rightarrow \langle \mathsf{Int} \rightarrow \mathsf{Int} \rangle$. This type says that power takes an integer and returns code. That code, when run, is a function from integers to integers. More efficient specialisers are possible. This program can be used as follows.

$$\mathsf{power}\ 2 \quad \twoheadrightarrow \quad \langle \lambda a.a \times ((\lambda b.b \times ((\lambda c.1)b))a) \rangle$$

The next example is *lifting* [6], which plays a vital role in meta-programming. Call a type α *basic* if it does not contain the function space constructor, i.e. if it has no subterms of the form $\beta \rightarrow \beta'$. The purpose of lifting is to take an arbitrary value V of basic type

α, and convert (lift) it to code $\langle V\rangle$ of type $\langle\alpha\rangle$. Note that we cannot simply write $\lambda x.\langle x\rangle$ because modal variables (i.e. variables free in code) cannot be λ-abstracted. For $\alpha = \mathsf{Int}$ the function is defined as follows:

$$\mathsf{lift}_{\mathsf{Int}} \quad \overset{def}{=} \quad \mu g.\lambda n^{\mathsf{Int}}.\texttt{if}\ n \leq 0\ \texttt{then}\ \langle 0\rangle\ \texttt{else}\ \texttt{let}\ \langle x\rangle = g(n-1)\ \texttt{in}\ \langle x+1\rangle.$$

Note that $\mathsf{lift}_{\mathsf{Int}}\,3$ evaluates to $\langle 0+1+1+1\rangle$, not $\langle 3\rangle$. In more expressive meta-programming languages such as Converge the corresponding program would evaluate to $\langle 3\rangle$, which is more efficient, although $\langle 0+1+1+1\rangle$ and $\langle 3\rangle$ are observationally indistinguishable.

Lifting easily extended to Unit and Bool, but not to function types. For basic types $\langle\alpha\rangle$ we can use the following definition:

$$\mathsf{lift}_{\langle\alpha\rangle} \overset{def}{=} \lambda x^{\langle\alpha\rangle}.\texttt{let}\ \langle a\rangle = x\ \texttt{in}\ \langle\langle a\rangle\rangle$$

PCF$_{\text{DP}}$ can implement a function of type $\langle\alpha\rangle \to \alpha$ for running code [6]:

$$\mathsf{eval} \overset{def}{=} \lambda x^{\langle\alpha\rangle}.\texttt{let}\ \langle y\rangle = x\ \texttt{in}\ y.$$

3 A Logic for Total Correctness

This section defines the syntax and semantics of the logic. Our logic is a Hoare logic with pre- and post-conditions in the tradition of logics for ML-like languages [1, 2, 11, 12]. *Expressions*, ranged over by $e, e', \ldots$ and *formulae* $A, B, \ldots$ of the logic are given by the grammar below, using the types and variables of PCF.

$$\begin{array}{lcl} e & ::= & \mathsf{c} \mid x \mid \mathsf{op}(\tilde{e}) \\ A & ::= & e = e' \mid \neg A \mid A \wedge B \mid \forall x^{\alpha}.A \mid u \bullet e = m\{A\} \mid u = \langle m\rangle\{A\} \end{array}$$

The logical language is based on standard first-order logic with equality. Other quantifiers and propositional connectives like $\supset$ (implication) are defined by de Morgan duality. Quantifiers range over values of appropriate type. Constants c and operations op are those of Section 2.

The proposed logic extends the logic for PCF of [9–11] with a new *code evaluation* primitive $u = \langle m\rangle\{A\}$. It says that u, which must be of type $\langle\alpha\rangle$, denotes (up to contextual congruence) a quasi-quoted program $\langle M\rangle$, such that whenever M is executed, it converges to a value; if that value is denoted by m then A makes a true statement about that value. We recall from [9–11] that $u \bullet e = m\{A\}$ says that (assuming u has function type) u denotes a function, which, when fed with the value denoted by e, terminates and yields another value. If we name this latter value m, A holds. We call the variable m in $u \bullet e = m\{A\}$ and $u = \langle m\rangle\{A\}$ an *anchor*. The anchor is a bound variable with scope A. The *free variables* of e and A, written $\mathsf{fv}(e)$ and $\mathsf{fv}(A)$, respectively, are defined as usual noting that $\mathsf{fv}(u = \langle m\rangle\{A\}) \overset{def}{=} (\mathsf{fv}(A) \setminus \{m\}) \cup \{u\}$. We use the following abbreviations: $A^{\text{-}x}$ indicates that $x \notin \mathsf{fv}(A)$ while $e \Downarrow$ means $\exists x^{\alpha}.e = x$, assuming that e has type α. We let $m = \langle e\rangle$ be short for $m = \langle x\rangle\{x = e\}$ where x is fresh, $m \bullet e = e'$ abbreviates $m \bullet e = x\{x = e'\}$ where x is fresh. We often omit typing annotations in expressions and formulae.

Judgements (for total correctness) are of the form $\{A\}\ M :_m \{B\}$. The variable m is the *anchor* of the judgement, is a bound variable with scope B, and not modal. The judgement is to be understood as follows: if A holds, then M terminates to a value, and if we denote that value m, then B holds. If a variable x occurs freely in A or in B, but not in M, then x is an *auxiliary variable* of the judgement $\{A\}\ M :_m \{B\}$. With environments as in Section 2, the *typing judgements* for expressions and formulae are $\Gamma;\Delta \vdash e : \alpha$ and $\Gamma;\Delta \vdash A$, respectively. The typing rules are given in Appendix A, together with the typing of judgements. The anchor in $u = \langle m \rangle\{A\}$ is modal, while it is not modal in $u \bullet e = m\{A\}$.

The logic has the following noteworthy features. (1) Quantification is not directly available on modal variables. (2) Equality is possible between modal and non-modal variables. The restriction on quantification makes the logic weaker for modal variables than first-order logic. Note that if x is modal in A we can form $\forall y.(x = y \supset A)$, using an equation between a modal and a non-modal variable. Quantification over all variables is easily defined by extending the grammar with a *modal quantifier* which ranges over arbitrary programs, not just values:

$$A \quad ::= \quad \ldots \mid \forall x^{\square\alpha}.A$$

For $\forall x^{\square\alpha}.A$ to be well-formed, x must be modal and of type α in A. The existential modal quantifier is given by de Morgan duality. Modal quantification is used only in the metalogical results of Section 5. We sometimes drop type annotations in quantifiers, e.g. writing $\forall x.A$. This shorthand will *never* be used for modal quantifiers. We abbreviate modal quantification to $\forall x^{\square}.A$. *From now on, we assume all occurring programs, expressions, formulae and judgements to be well-typed.*

Examples of Assertions & Judgements. We continue with a few simple examples to motivate the use of our logic.

- The assertion $m = \langle 3 \rangle$, which is short for $m = \langle x \rangle\{x = 3\}$ says that m denotes code which, when executed, will evaluate to 3. It can be used to assert on the program $\langle 1+2 \rangle$ as follows: $\{\top\}\ \langle 1+2 \rangle :_m \{m = \langle 3 \rangle\}$
- Let Ω_α be a non-terminating program of type α (we usually drop the type subscript). When we quasi-quote Ω, the judgement $\{\top\}\ \langle \Omega \rangle :_m \{\top\}$ says (*qua* precondition) that $\langle \Omega \rangle$ is a terminating program. Indeed, that is the strongest statement we can make about $\langle \Omega \rangle$ in a logic for total correctness.
- The assertion $\forall x^{\mathsf{Int}}.m \bullet x = y\{y = \langle x \rangle\}$ says that m denotes a terminating function which receives an integer and returns code which evaluates to that integer. Later we use this assertion when reasoning about $\mathsf{lift}_{\mathsf{Int}}$ which has the following specification.

$$\{\top\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.n \geq 0 \supset u \bullet n = m\{m = \langle n \rangle\}\}$$

- The formula $A_u \overset{def}{=} \forall n^{\mathsf{Int}} \geq 0.\exists f^{\mathsf{Int}\to\mathsf{Int}}.(u \bullet n = \langle f \rangle \wedge \forall x^{\mathsf{Int}}.f \bullet x = x^n)$ says that u denotes a function which receives an integer n as argument, to return code which when evaluated and fed another integer x, computes the power x^n, provided $n \geq 0$. We can show that $\{\top\}\ \mathsf{power} :_u \{A_u\}$ and $\{A_u\}\ u\ 7 :_r \{r = \langle f \rangle\{\forall x.f \bullet x = x^7\}\}$.
- The formula $\forall x^{\langle\alpha\rangle} y^{\alpha}.(x = \langle y \rangle \supset u \bullet x = y)$ can be used to specify the evaluation function from Section 2: $\{\top\}\ \mathsf{eval} :_u \{\forall x^{\langle\alpha\rangle} y^{\alpha}.(x = \langle y \rangle \supset u \bullet x = y)\}$

Models and the Satisfaction Relation. This section formally presents the semantics of our logic. We begin with the notion of model. The key difference from the models of [11] is that modal variables denote possibly non-terminating programs.

$(e1)$	$x\bullet y = z\{A\} \wedge x\bullet y = z\{B\}$	$\equiv$	$x\bullet y = z\{A \wedge B\}$	
$(e2)$	$x\bullet y = z\{\neg A\}$	$\supset$	$\neg x\bullet y = z\{A\}$	
$(e3)$	$x\bullet y = z\{A\} \wedge \neg x\bullet y = z\{B\}$	$\equiv$	$x\bullet y = z\{A \wedge \neg B\}$	
$(e4)$	$x\bullet y = z\{A \wedge B\}$	$\equiv$	$A \wedge x\bullet y = z\{B\}$	$z \notin \mathsf{fv}(A)$
$(e5)$	$x\bullet y = z\{\forall a^{\alpha}.A\}$	$\equiv$	$\forall a^{\alpha}.x\bullet y = z\{A\}$	$a \neq x,y,z$
$(e6)$	$(A \supset B) \wedge x\bullet y = z\{A\}$	$\supset$	$x\bullet y = z\{B\}$	$z \notin \mathsf{fv}(A,B)$
(ext)	$x = y$	$\equiv$	$\mathsf{Ext}(xy)$	

Fig. 2. Key total correctness axioms for PCF_{DP}.

Let Γ,Δ be two contexts with disjoint domains (the idea is that Δ is modal while Γ is not). A *model* of type $\Gamma;\Delta$ is a pair (ξ,σ) where ξ is a map from $\mathsf{dom}(\Gamma)$ to closed *values* such that $\vdash \xi(x) : \Gamma(x)$; σ is a map from $\mathsf{dom}(\Delta)$ to closed *programs* $\vdash \sigma(x) : \Delta(x)$. We also write $(\xi,\sigma)^{\Gamma;\Delta}$ to indicate that (ξ,σ) is a model of type $\Gamma;\Delta$. We write $\xi \cdot x : V$ for $\xi \cup \{(x,V)\}$ assuming that $x \notin \mathsf{dom}(\xi)$, and likewise for $\sigma \cdot x : VM$. We can now present the semantics of expressions. Let $\Gamma;\Delta \vdash e : \alpha$ and assume that (ξ,σ) is a $\Gamma;\Delta$-model, we define $[\![e]\!]_{(\xi,\sigma)}$ by the following inductive clauses. $[\![\mathsf{c}]\!]_{(\xi,\sigma)} \stackrel{def}{=} \mathsf{c}$, $[\![\mathsf{op}(\tilde{e})]\!]_{(\xi,\sigma)} \stackrel{def}{=} \mathsf{op}([\![\tilde{e}]\!]_{(\xi,\sigma)})$, $[\![x]\!]_{(\xi,\sigma)} \stackrel{def}{=} (\xi \cup \sigma)(x)$. The satisfaction relation for formulae has the following shape. Let $\Gamma;\Delta \vdash A$ and assume that (ξ,σ) is a $\Gamma;\Delta$-model. We define $(\xi,\sigma) \models A$ as usual with two extensions.

- $(\xi,\sigma) \models e = e'$ iff $[\![e]\!]_{(\xi,\sigma)} \simeq [\![e']\!]_{(\xi,\sigma)}$.
- $(\xi,\sigma) \models \neg A$ iff $(\xi,\sigma) \not\models A$.
- $(\xi,\sigma) \models A \wedge B$ iff $(\xi,\sigma) \models A$ and $(\xi,\sigma) \models B$.
- $(\xi,\sigma) \models \forall x^{\alpha}.A$ iff for all closed values V of type α: $(\xi \cdot x : V,\sigma) \models A$.
- $(\xi,\sigma) \models u \bullet e = x\{A\}$ iff $([\![u]\!]_{(\xi,\sigma)}[\![e]\!]_{(\xi,\sigma)}) \Downarrow V$ and $(\xi \cdot x : V,\sigma) \models A$.
- $(\xi,\sigma) \models u = \langle m\rangle\{A\}$ iff $[\![u]\!]_{(\xi,\sigma)} \Downarrow \langle M\rangle$, $M \Downarrow V$ and $(\xi,\sigma \cdot m : V) \models A$.

To define the semantics of judgements, we need to say what it means to apply a model (ξ,σ) to a program M, written $M(\xi,\sigma)$. That is defined as usual, e.g. $x(\xi,\sigma) \stackrel{def}{=} (\xi \cup \sigma)(x)$ and $(MN)(\xi,\sigma) \stackrel{def}{=} M(\xi,\sigma)N(\xi,\sigma)$.

The *satisfaction relation* $\models \{A\}\, M :_m \{B\}$ is given next. Let $\Gamma;\Delta;\alpha \vdash \{A\}\, M :_m \{B\}$. Then

$$\models \{A\}\, M :_m \{B\} \quad \text{iff} \quad \forall(\xi,\sigma)^{\Gamma;\Delta}.(\xi,\sigma) \models A \;\supset \exists V.(M(\xi,\sigma) \Downarrow V \;\wedge (\xi \cdot m : V,\sigma) \models B).$$

This is the standard notion of total correctness, adapted to the present logic.

Axioms. This section introduces the key axioms of the logic. All axioms of [9–11] remain valid. We add axioms for $x = \langle y\rangle\{A\}$. Tacitly, we assume typability of all axioms. Some key axioms are given in Figure 2, more precisely, the axioms are the universal closure of the formulae presented in Figure 2. The presentation of axioms uses the following abbreviation: $\mathsf{Ext}_{\mathsf{q}}(xy)$ stands for $\forall a.(x = \langle z\rangle\{z = a\} \equiv y = \langle z\rangle\{z = a\})$.
Axiom $(q1)$ says that if the quasi-quote denoted by x makes A true (assuming the program in that quasi-quote is denoted by y), and in the same way makes B true, then it also makes $A \wedge B$ true, and vice versa. Axiom $(q2)$ says that if the quasi-quote denoted by x contains

$$\frac{}{\{A[x/m] \wedge x \Downarrow\}\ x :_m \{A\}}\ \textsc{Var} \qquad \frac{}{\{A[\mathsf{c}/m]\}\ \mathtt{c} :_m \{A\}}\ \textsc{Const} \qquad \frac{\{A^{\text{-}g}\}\ M :_u \{B\}}{\{A\}\ \mu g.M :_u \{B[u/g]\}}\ \textsc{Rec}$$

$$\frac{\{A^{\text{-}x} \wedge B\}\ M :_m \{C\}}{\{A\}\ \lambda x^{\alpha}.M :_u \{\forall x.(B \supset u \bullet x = m\{C\})\}}\ \textsc{Abs} \qquad \frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{C[m+n/u]\}}{\{A\}\ M+N :_u \{C\}}\ \textsc{Add}$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B[b_i/m]\}\ N_i :_u \{C\} \quad b_1 = \mathsf{t}, b_2 = \mathsf{f} \quad i = 1,2}{\{A\}\ \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u \{C\}}\ \textsc{If}$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{m \bullet n = u\{C\}\}}{\{A\}\ MN :_u \{C\}}\ \textsc{App} \qquad \frac{\{A\}\ M :_m \{B\}}{\{\mathsf{T}\}\ \langle M \rangle :_u \{A \supset u = \langle m \rangle \{B\}\}}\ \textsc{Quote}$$

$$\frac{\{A\}\ M :_m \{B^{\text{-}mx} \supset m = \langle x \rangle \{C^{\text{-}m}\}\} \quad \{B \supset C\}\ N :_u \{D^{\text{-}mx}\}}{\{A\}\ \mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ N :_u \{D\}}\ \textsc{Unquote}$$

Fig. 3. Key $\textsc{Pcf}_{\textsc{DP}}$ inference rules for total correctness.

a terminating program, denoted by y, and makes $\neg A$ true, then it cannot be the case that under the same conditions A holds. The reverse implication is false, because $\neg x = \langle y \rangle \{A\}$ is also true when x denotes a quasi-quote whose contained program is diverging. Next is $(q3)$: $x = \langle y \rangle \{A\}$ says in particular that x denotes a quasi-quote containing a terminating program, so $\neg x = \langle y \rangle \{B\}$ can only be true because B is false. Axioms $(q4, q5)$ let us move formulae and quantifiers in and out of code-evaluation formulae, as long as the anchor is not inappropriately affected. Axiom $(q6)$ enables us to weaken a code-evaluation formula. The code-extensionality axiom (ext_q) formalises what it means for two quasi-quotes to be equal: they must contain observationally indistinguishable code.

Rules. Key rules of inference can be found in Figure 3. We write $\vdash \{A\}\ M :_m \{B\}$ to indicate that $\{A\}\ M :_m \{B\}$ is derivable using these rules. Rules make use of capture-free syntactic substitutions $e[e'/x]$, $A[e/x]$ which is straightforward, except that in $e[e'/x]$ and $A[e/x]$, x must be non-modal. Structural rules like Hoare's rule of consequence, are unchanged from [9–11] and used without further comment. The rules in Figure 3 are standard and also unchanged from [9–11] with three significant exceptions, explained next.

[$\textsc{Var}$] adds $x \Downarrow$, i.e. $\exists a.x = a$ in the precondition. By construction of our models, $x \Downarrow$ is trivially true if x is non-modal. If x is modal, the situation is different because x may denote a non-terminating program. In this case $x \Downarrow$ constrains x so that it really denotes a value, as is required in a total correctness logic.

[$\textsc{Quote}$] says that $\langle M \rangle$ always terminates (because the conclusion's precondition is simply T). Moreover, if u denotes the result of evaluating $\langle M \rangle$, i.e. $\langle M \rangle$ itself, then, assuming A holds (i.e., given the premise, if M terminates), u contains a terminating program, denoted m, making B true. Clearly, in a logic of total correctness, if M is not a terminating program, A will be equivalent to F, in which case, [$\textsc{Quote}$] does not make a non-trivial assertion about $\langle M \rangle$ beyond stating that $\langle M \rangle$ terminates.

[$\textsc{Unquote}$] is similar to the usual rule for $\mathtt{let}\ x = M\ \mathtt{in}\ N$ which is easily derivable:

$$\frac{\{A\}\ M :_x \{B\} \quad \{B\}\ N :_u \{C\}}{\{A\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C\}}\ \textsc{Let}$$

Rules for `let` $\langle x\rangle = M$ `in` N are more difficult because a quasi-quote always terminates, but the code it contains may not. Moreover, even if M evaluates to a quasi-quote containing a divergent program, the overall expression may still terminate, because N uses the destructed quasi-quote in a certain manner. Here is an example:

$$\texttt{let}\ \langle x\rangle = \langle\Omega\rangle\ \texttt{in}\ ((\lambda ab.a)\ 7\ (\lambda().x)).$$

[UNQUOTE] deals with this complication in the following way. Assume $\{A\}\ M :_m \{B \supset m = \langle x\rangle\{C\}\}$ holds. If M evaluates to a quasi-quote containing a divergent program, B would be equivalent to F. The rule uses $B \supset C$ in the right premise, where x is now a free variable, hence also constrained by C. If B is equivalent to F, the right precondition is T, i.e. contains no information, and M's termination behaviour cannot depend on x, i.e. N must use whatever x denotes in a way that makes the termination or otherwise of N independent of x. Apart from this complication, the rule is similar to [LET].

4 Reasoning Examples

We now put our logic to use by reasoning about some of the programs introduced in Section 2. The derivations use the abbreviations of Section 3 and omit many steps that are not to do with meta-programming. Several reduction steps are justified by the following two standard structural rules omitted from Figure 3.

$$\frac{A \supset A' \quad \{A'\}\ M :_u \{B'\} \quad B' \supset B}{\{A\}\ M :_u \{B\}}\ \text{CONSEQ} \qquad \frac{\{A\}\ M :_m \{B \supset C\}}{\{A \wedge B\}\ M :_m \{C\}}\ \supset\text{-}\wedge$$

Example 1. We begin with the simple program $\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$. The derivation is straightforward.

1	$\{\mathsf{T}\}\ 1+2 :_a \{a = 3\}$	
2	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{\mathsf{T} \supset m = \langle a\rangle\{a = 3\}\}$	QUOTE, *1*
3	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$	CONSEQ, *2*

Example 2. This example deals with the code of a non-terminating program. We derive $\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$. This is the strongest total correctness assertion about $\langle\Omega\rangle$. In the proof, we assume that $\{\mathsf{F}\}\ \Omega :_a \{\mathsf{T}\}$ is derivable, which is easy to show.

1	$\{\mathsf{F}\}\ \Omega :_a \{\mathsf{T}\}$	
2	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{F} \supset m = \langle a\rangle\{\mathsf{T}\}\}$	QUOTE, *1*
3	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$	CONSEQ, *2*

Example 3. The third example destructs a quasi-quote and then injects the resulting program into another quasi-quote.

$$\{\mathsf{T}\}\ \texttt{let}\ \langle x\rangle = \langle 1+2\rangle\ \texttt{in}\ \langle x+3\rangle :_m \{m = \langle 6\rangle\}$$

We derive the assertion in small steps to demonstrate how to apply our logical rules.

1	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m=\langle 3\rangle\}$	*Ex. 1*
2	$\{(a=3)[x/a]\wedge x\Downarrow\}\ x :_a \{a=3\}$	VAR
3	$\{x=3\}\ x :_a \{a=3\}$	CONSEQ, *2*
4	$\{\mathsf{T}\}\ 3 :_b \{b=3\}$	CONST, CONSEQ
5	$\{a=3\}\ 3 :_b \{a=3\wedge b=3\}$	INVAR, *4*
6	$\{a=3\}\ 3 :_b \{(c=6)[a+b/c]\}$	CONSEQ, *5*
7	$\{x=3\}\ x+3 :_c \{c=6\}$	ADD, *3, 6*
8	$\{\mathsf{T}\}\ \langle x+3\rangle :_u \{x=3\supset u=\langle c\rangle\{c=6\}\}$	QUOTE, *7*
9	$\{x=3\}\ \langle x+3\rangle :_u \{u=\langle c\rangle\{c=6\}\}$	$\supset$-$\wedge$, *8*
10	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{\mathsf{T}\supset m=\langle x\rangle\{x=3\}\}$	CONSEQ, *1*
11	$\{\mathsf{T}\supset x=3\}\ \langle x+3\rangle :_u \{u=\langle 6\rangle\}$	CONSEQ, *9*
12	$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle=\langle 1+2\rangle\ \mathtt{in}\ \langle x+3\rangle :_u \{u=\langle 6\rangle\}$	UNQUOTE, *10, 11*

Example 4. Now we show that destructing a quasi-quote containing a non-terminating program, and then not using that program still leads to a terminating program. This reflects the operational semantics in Section 2.

$$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle=\langle\Omega\rangle\ \mathtt{in}\ \langle 1+2\rangle :_m \{m=\langle 3\rangle\}$$

The derivation follows.

1	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$	*Ex. 2*
2	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{F}\supset m=\langle a\rangle\{\mathsf{T}\}\}$	CONSEQ, *1*
3	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m=\langle 3\rangle\}$	*Ex. 1*
4	$\{\mathsf{F}\supset\mathsf{T}\}\ \langle 1+2\rangle :_m \{m=\langle 3\rangle\}$	CONSEQ, *3*
5	$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle=\langle\Omega\rangle\ \mathtt{in}\ \langle 1+2\rangle :_m \{m=\langle 3\rangle\}$	UNQUOTE, *2, 4*

The examples below make use of the following convenient forms of the recursion rule and [UNQUOTE]. Both are easily derived.

$$\frac{\{A^{\text{-}gn}\wedge\forall 0\le i<n.B[i/n][g/u]\}\ \lambda x.M :_u \{B^{\text{-}g}\}}{\{A\}\ \mu g.\lambda x.M :_u \{\forall n\ge 0.B\}}\ \scriptstyle\mathrm{REC'}\qquad \frac{\{A\}\ M :_m \{\mathsf{T}\}\quad \{\mathsf{T}\}\ N :_u \{B\}}{\{A\}\ \mathtt{let}\ \langle x\rangle=M\ \mathtt{in}\ N :_u \{B\}}\ \scriptstyle\mathrm{UQ}$$

Example 5. This example extract a non-terminating program from a quasi-quote, and injects it into a new quasi-quote. Our total-correctness logic cannot say anything non-trivial about the resulting quasi-quote (cf. Example 2):

$$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle=\langle\Omega\rangle\ \mathtt{in}\ \langle x\rangle :_u \{\mathsf{T}\}$$

The derivation is straightforward.

1	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$	*Ex. 2*
2	$\{\mathsf{F}[x/a] \wedge x \Downarrow\}\ x :_a \{\mathsf{F}\}$	VAR
3	$\{\mathsf{F}\}\ x :_a \{\mathsf{T}\}$	CONSEQ, *2*
4	$\{\mathsf{T}\}\ \langle x\rangle :_u \{\mathsf{F} \supset u = \langle a\rangle\{\mathsf{T}\}\}$	QUOTE, *3*
5	$\{\mathsf{T}\}\ \texttt{let}\ \langle x\rangle = \langle\Omega\rangle\ \texttt{in}\ \langle x\rangle :_u \{\mathsf{T}\}$	UQ, *1, 4*

Example 6. Now we reason about $\mathsf{lift}_{\mathsf{Int}}$ from Section 3. In the proof we assume that i, n range over non-negative integers. Let $A^u_n \stackrel{def}{=} u \bullet n = m\{m = \langle n\rangle\}$. We are going to establish the following assertion from Section 3: $\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.A^u_n\}$. We set $C \stackrel{def}{=} i \leq n \wedge \forall j < n.A^g_j$, $D \stackrel{def}{=} i > 0 \wedge \forall r.(0 \leq r < n \supset g \bullet r = m\{m = \langle r\rangle\})$ and $P \stackrel{def}{=} \texttt{let}\ \langle x\rangle = g(i-1)\ \texttt{in}\ \langle x+1\rangle$.

1	$\{C\}\ i \leq 0 :_b \{C \wedge (b = \mathsf{t} \equiv i \leq 0)\}$	
2	$\{\mathsf{T}\}\ \langle 0\rangle :_m \{m = \langle 0\rangle\}$	*Like Ex. 1*

3	$\{i = 0\}\ \langle 0\rangle :_m \{m = \langle i\rangle\}$	INVAR, CONSEQ, *2*
4	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{t}/b]\}\ \langle 0\rangle :_m \{m = \langle i\rangle\}$	CONSEQ, *3*
5	$\{x = i - 1\}\ \langle x+1\rangle :_m \{m = \langle i\rangle\}$	*Like Ex. 3*
6	$\{\mathsf{T} \supset x = i - 1\}\ \langle x+1\rangle :_m \{m = \langle i\rangle\}$	CONSEQ, *5*
7	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{f}/b]\}\ g :_s \{D\}$	VAR
8	$\{D\}\ i - 1 :_r \{g \bullet r = t\{t = \langle i-1\rangle\}\}$	
9	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{f}/b]\}\ g(i-1) :_t \{t = \langle i-1\rangle\}$	APP, *7, 8*
10	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{f}/b]\}\ P :_m \{m = \langle i\rangle\}$	UNQUOTE, CONSEQ, *6, 9*
11	$\{C\}\ \texttt{if}\ i \leq 0\ \texttt{then}\ \langle 0\rangle\ \texttt{else}\ P :_m \{m = \langle i\rangle\}$	IF, *4, 10*
12	$\{\mathsf{T}\}\ \lambda i.\texttt{if}\ i \leq 0\ \texttt{then}\ \langle 0\rangle\ \texttt{else}\ P :_u \{\forall i.(C \supset A^u_i)\}$	ABS, *11*
13	$\{\forall j < n.A^g_j\}\ \lambda i.\texttt{if}\ i \leq 0\ \texttt{then}\ \langle 0\rangle\ \texttt{else}\ P :_u \{\forall i \leq n.A^u_i\}$	CONSEQ $\supset$-$\wedge$, *12*
14	$\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.\forall i \leq n.A^u_n\}$	REC', *13*
15	$\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.A^u_n\}$	CONSEQ, *14*

Example 7. We close this section by reasoning about the staged power function from Section 2. Assuming that i, j, k, n range over non-negative integers, we define $B^u_n \stackrel{def}{=} u \bullet n = m\{m = \langle y\rangle\{\forall j.y \bullet j = j^n\}\}$. In the derivation, we provide less detail than in previous

proofs for readability.

1	$C \overset{def}{=} n \le k \wedge \forall i < k.B_i^p \qquad D \overset{def}{=} C \wedge (b = \mathtt{t} \wedge n \le 0)$	
2	$P \overset{def}{=} \mathtt{let}\ \langle q\rangle = p(n-1)\ \mathtt{in}\ \langle \lambda x.x \times (q\ x)\rangle$	
3	$\{C\}\ n \le 0 :_b \{D\}$	
4	$\{D[\mathtt{t}/b]\}\ \langle \lambda x.1\rangle :_m \{m = \langle y\rangle\{\forall j.y \bullet j = j^n\}\}$	*Like prev. examples*
5	$\{D[\mathtt{f}/b]\}\ p(n-1) :_r \{\top \supset r = \langle q\rangle\{\forall j.q \bullet j = j^{n-1}\}\}$	*Like Ex. 6*
6	$\{\top \supset \forall j.q \bullet j = j^{n-1}\}\ \langle \lambda x.x \times (q\ x)\rangle :_m \{m = \langle y\rangle\{\forall j.y \bullet j = j^n\}\}$	*Like Ex. 6*
7	$\{D[\mathtt{f}/b]\}\ P :_m \{m = \langle y\rangle\{\forall j.y \bullet j = j^n\}\}$	UNQUOTE, *5, 6*
8	$\{C\}\ \mathtt{if}\ n \le 0\ \mathtt{then}\ \langle \lambda x.1\rangle\ \mathtt{else}\ P :_m \{m = \langle y\rangle\{\forall j.y \bullet j = j^n\}\}$	IF, *7*
9	$\{\top\}\ \lambda n.\mathtt{if}\ n \le 0\ \mathtt{then}\ \langle \lambda x.1\rangle\ \mathtt{else}\ P :_u \{\forall n \le k.((\forall i < k.B_i^p) \supset B_n^u)\}$	ABS, *8*
10	$\{\forall i < k.B_i^p\}\ \lambda n.\mathtt{if}\ n \le 0\ \mathtt{then}\ \langle \lambda x.1\rangle\ \mathtt{else}\ P :_u \{\forall n \le k.B_n^u\}$	CONSEQ, *9*
11	$\{\top\}\ \mathsf{power} :_u \{\forall k.\forall n \le k.B_n^u\}$	REC', *10*
12	$\{\top\}\ \mathsf{power} :_u \{\forall n.B_n^u\}$	CONSEQ, *11*

5 Completeness

This section answers three important metalogical questions about the logic for total correctness introduced in Section 3.

- Is the logic *relatively complete* in the sense of Cook [5]? This question asks if $\models \{A\}\ M :_m \{B\}$ implies $\vdash \{A\}\ M :_m \{B\}$ for all appropriate A, B. Relative completeness means that the logic can syntactically derive all semantically true assertions, and reasoning about programs does not need to concern itself with models. We can always rely on just syntactic rules to derive an assertion (assuming an oracle for Peano arithmetic).

$$\frac{x \text{ non-modal}}{\{\top\}\ x :_m \{x = m\}}\ \textsc{Var} \qquad \frac{x \text{ modal}}{\{x \Downarrow\}\ x :_m \{x = m\}}\ \textsc{Var}_m \qquad \frac{\{A\}\ M :_m \{B\}}{\{\top\}\ \langle M\rangle :_u \{A \supset u = \langle m\rangle\{B\}\}}\ \textsc{Quote}$$

$$\frac{\{A_1\}\ M :_m \{B_1\} \quad \{A_2\}\ N :_u \{B_2\}}{\begin{array}{c}\{A_1 \wedge ((\forall m x^{\Box}.A_2) \vee \forall m.(B_1 \supset m = \langle x\rangle\{A_2\}))\} \\ \mathtt{let}\ \langle x\rangle = M\ \mathtt{in}\ N :_u \\ \{\exists m x^{\Box}.((m = \langle \cdot \rangle \supset m = \langle x\rangle) \wedge B_1 \wedge B_2)\}\end{array}}\ \textsc{Uq}$$

Fig. 4. Key inference system for TCAPs, where $m = \langle \cdot \rangle$ is short for $m = \langle z\rangle\{\top\}$.

- Is the logic *observationally complete* [10]? The second question investigates if the program logic makes the same distinctions as the observational congruence. In other words, does $M \simeq N$ hold iff for all suitably typed A, B: $\{A\}\ M :_m \{B\}$ iff $\{A\}\ N :_m$

$\{B\}$? Observational completeness means that the operational semantics (given by the contextual congruence) and the axiomatic semantics given by logic cohere with each other.

– If a logic is observationally complete, we may ask: given a program M, can we find, by induction on the syntax of M, *characteristic formulae* A, B such that (1) $\models \{A\}\ M :_m \{B\}$ and (2) for all programs N: $M \simeq N$ iff $\models \{A\}\ N :_m \{B\}$? If characteristic formulae always exist, the semantics of each program can be obtained and expressed finitely in the logic, and we call the logic *descriptively complete* [10].

Following [3, 10, 21], we answer all questions in the affirmative.

Characteristic Formulae. Program logics reason about program properties denoted by pairs of formulae. But what are program properties? We cannot simply say program properties are subsets of programs, because there are uncountably many such subsets, yet only countably many pairs of formulae. To obtain a notion of program property that is appropriate for a logic of total correctness, we note that such logics cannot express that a program diverges. More generally, if $\models \{A\}\ M :_m \{B\}$ and $M \sqsubseteq N$ (where $\sqsubseteq$ is the contextual pre-congruence from Section 2), then also $\models \{A\}\ N :_m \{B\}$. Thus pairs A, B talk about *upwards-closed* sets of programs. A set S of programs is upwards-closed if $M \in S$ and $M \sqsubseteq N$ together imply that $N \in S$. It can be shown that each upwards closed set of PCF$_{\text{DP}}$-terms has a unique least element up-to $\simeq$. Thus each upwards-closed set has a distinguished member, is its least element. Consequently a pair A, B is a characteristic assertion pair for M (at m) if M is the *least* program w.r.t. $\sqsubseteq$ such that $\models \{A\}\ M :_m \{B\}$, leading to the following key definition.

Definition. A pair (A, B) is a *total characteristic assertion pair*, or *TCAP*, *of M at u*, if the following conditions hold (in each clause we assume well-typedness).

1. (soundness) $\models \{A\}\ M :_u \{B\}$.
2. (MTC, minimal terminating condition) For all models (ξ, σ), $M(\xi, \sigma) \Downarrow$ if and only if $(\xi, \sigma) \models A$.
3. (closure) If $\models \{E\}\ N :_u \{B\}$ and $E \supset A$, then for all (ξ, σ): $(\xi, \sigma) \models E$ implies $M(\xi, \sigma) \sqsubseteq N(\xi, \sigma)$.

A TCAP of M denotes a set of programs whose minimum element is M, and in that sense characterises that behaviour uniquely up to $\sqsubseteq$.

Descriptive Completeness. The main tool in answering the three questions posed above is the inference system for TCAPs, of which the key rules are given in Figure 4. The remaining rules are unchanged from [10]. We write $\vdash^{\mathsf{tcap}} \{A\}\ M :_u \{B\}$ to indicate that $\{A\}\ M :_u \{B\}$ is derivable in that new inference system. It is obvious that TCAPs can be derived mechanically from programs – no invariants for recursion have to be supplied manually.

The premise of [UNQUOTE] in Figure 4 uses modal quantification. This is the only use of the modal quantifier. The semantics is: $(\xi, \sigma) \models \forall x^{\Box\alpha}.A$ iff for all closed programs M of type α: $(\xi, \sigma \cdot x : M) \models A$. Syntactic reasoning with modal quantifiers needs a few straightforward quantifier axioms beyond those of first-order logic and those of Figure 2, for example $\neg\forall x^{\Box}.x \Downarrow$, and $\neg\forall x^{\Box}.\neg x \Downarrow$. An interesting open question is whether modal quantification can be avoided altogether in constructing TCAPs.

Theorem 1.

1. *(descriptive completeness for total correctness)* Assume $\Gamma;\Delta \vdash M : \alpha$. Then $\vdash^{\mathsf{tcap}} \{A\}\, M :_u \{B\}$ implies (A,B) is a TCAP of M at u.
2. (observational completeness) $M \simeq N$ if and only if, for each A and B, we have $\models \{A\}\, M :_u \{B\}$ iff $\models \{A\}\, N :_u \{B\}$.
3. (relative completeness) Let B be upward-closed at u. Then $\models \{A\}\, M :_u \{B\}$ implies $\vdash \{A\}\, M :_u \{B\}$.

6 Conclusion

We have proposed the first program logic for a meta-programming language, and established key metalogical properties like completeness and the correspondence between axiomatic and operational semantics. We are not aware of previous work on program logics for meta-programming. Instead, typing systems for statically enforcing program properties have been investigated. We discuss the two systems with the most expressive typing systems, Ωmega [15] and Concoqtion [7]. Both use indexed typed to achieve expressivity. Ωmega is a CBV variant of Haskell with generalised algebraic datatypes (GADTs) and an extensible kind system. In Ωmega, GADTs can express easily datatypes representing object-programs, whose meta-level types encode the object-level types of the programs represented. Tagless interpreters can directly be expressed and typed for these object programs. Ωmega is expressive enough to encode the MetaML typing system together with a MetaML interpreter in a type-safe manner. Concoqtion is an extension of MetaOCaml and uses the term language of the theorem prover Coq to define index types, specify index operations, represent the properties of indexes and construct proofs. Basing indices on Coq terms opens all mathematical insight available as Coq libraries to use in typing meta-programs. Types in both languages are not as expressive with respect to properties of *meta-programs themselves*, as our logics, which capture exactly the observable properties. Nevertheless, program logic and type-theory are not mutually exclusive; on the contrary, reconciling both in the context of meta-programming is an important open problem.

The construction of our logics as extensions of well-understood logics for PCF indicates that logical treatment of meta-programming is mostly orthogonal to that of other language features. Hence [18] is an interesting target for generalising the techniques proposed here because it forms the basis of MetaOCaml, the most widely studied meta-programming language in the MetaML tradition. $\mathrm{PCF}_{\mathrm{DP}}$ and [18] are similar as meta-programming languages with the exception that the latter's typing system is substantially more permissive: even limited forms evaluation of *open* code is possible. We believe that a logical account of meta-programming with open code is a key challenge in bringing program logics to realistic meta-programming languages. A different challenge is to add state to $\mathrm{PCF}_{\mathrm{DP}}$ and extend the corresponding logics. We expect the logical treatment of state given in [2, 21] to extend smoothly to a meta-programming setting. The main issue is the question what typing system to use to type stateful meta-programming: the system used in MetaOCaml, based on [18], is unsound in the presence of state due to a form of scope extrusion. This problem is mitigated in MetaOCaml with dynamic type-checking. As an alternative to dynamic typing, the Java-like meta-programming language Mint [20] simply prohibits the *sharing* of state between different meta-programming stages, resulting in a statically sound typing system. We believe that both suggestions can be made to coexist with modern logics for higher-order state [2, 21], in the case of [20] easily so.

The relationship between PCF_{DP} and PCF, its non-meta-programming core, is also worth investigating. Pfenning and Davies proposed an embedding $\ulcorner\cdot\urcorner$ from PCF_{DP} into PCF, whose main clauses are given next.

$$\ulcorner\langle\alpha\rangle\urcorner \overset{def}{=} \mathsf{Unit} \to \ulcorner\alpha\urcorner$$
$$\ulcorner\langle M\rangle\urcorner \overset{def}{=} \lambda().\ulcorner M\urcorner$$
$$\ulcorner \mathtt{let}\ \langle x\rangle = M\ \mathtt{in}\ N\urcorner \overset{def}{=} \mathtt{let}\ x = \ulcorner M\urcorner\ \mathtt{in}\ \ulcorner N\urcorner[x()/x]$$

We believe that this embedding is fully-abstract, but proving full abstraction is non-trivial because translated PCF_{DP}-term have PCF-inhabitants which are not translations of PCF_{DP}-terms (e.g. $\lambda x^{\mathsf{Int}\to\mathsf{Int}}.\lambda().x$). A full abstraction proof might be useful in constructing a logically fully abstract embedding of the logic presented here into the simpler logic for PCF from [9, 11]. A logical full abstraction result [13] is an important step towards integrating logics for meta-programming with logics for the produced meta-programs.

In the light of this encoding one may ask why meta-programming languages are relevant at all: why not simply work with a non-meta-programming language and encodings?

- We believe that nice (i.e. fully abstract and compositional) encodings might exist for simple meta-programming languages like PCF_{DP} because PCF_{DP} lives at the low end of meta-programming expressivity. For even moderately more expressive languages like [18] no fully abstract encodings into simple λ-calculi are known.
- A second reason is to do with efficiency, one of the key reasons for using meta-programming: encodings are unlikely to be as efficient as proper meta-programming.
- Finally, programs written using powerful meta-programming primitives are more readable and hence more easily evolvable than equivalent programs written using encodings.

References

1. M. Berger. Program Logics for Sequential Higher-Order Control. In *Proc. FSEN*, pages 194–211, 2009.
2. M. Berger, K. Honda, and N. Yoshida. A Logical Analysis of Aliasing in Imperative Higher-Order Functions. *J. Funct. Program.*, 17(4-5):473–546, 2007.
3. M. Berger, K. Honda, and N. Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In *Proc. ICALP*, pages 99–111, 2008.
4. M. Berger and L. Tratt. Program Logics for Homogeneous Metaprogramming. Long version of the present paper, to appear.
5. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
6. R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.
7. S. Fogarty, E. Pašalić, J. Siek, and W. Taha. Concoqtion: Indexed Types Now! In *Proc. PEPM*, pages 112–121, 2007.
8. C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
9. K. Honda. From Process Logic to Program Logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
10. K. Honda, M. Berger, and N. Yoshida. Descriptive and Relative Completeness of Logics for Higher-Order Functions. In *Proc. ICALP*, pages 360–371, 2006.

11. K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP*, pages 191–202, 2004.
12. K. Honda, N. Yoshida, and M. Berger. An Observationally Complete Program Logic for Imperative Higher-Order Functions. In *LICS'05*, pages 270–279, 2005.
13. J. Longley and G. Plotkin. Logical Full Abstraction and PCF. In *Tbilisi Symposium on Logic, Language and Information*, CSLI, 1998.
14. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, 2002.
15. T. Sheard and N. Linger. Programming in Ωmega. In *Proc. Central European Functional Programming School*, pages 158–227, 2007.
16. T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proc. Haskell Workshop*, pages 1–16, 2002.
17. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1993.
18. W. Taha and M. F. Nielsen. Environment classifiers. In *Proc. POPL*, pages 26–37, 2003.
19. L. Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proc. DLS*, pages 49–64, Oct. 2005.
20. E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *Proc. PLDI*, 2010. To appear.
21. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. In *Proc. Fossacs*, LNCS, pages 361–377, 2007.

A Typing Rules for Expressions, Formulae and Assertions

The key typing rules for expressions, formulae and judgements are summarised in Figure 5 above.

$$\frac{(x,\alpha)\in\Gamma\cup\Delta}{\Gamma;\Delta\vdash x:\alpha}\qquad\frac{\Gamma;\Delta\vdash u:\alpha\to\beta\quad\Gamma;\Delta\vdash e:\alpha\quad\Gamma,m:\beta;\Delta\vdash A}{\Gamma;\Delta\vdash u\bullet e=m\{A\}}\qquad\frac{\Gamma;\Delta\vdash e:\alpha\quad\Gamma;\Delta\vdash e':\alpha}{\Gamma;\Delta\vdash e=e'}$$

$$\frac{\Gamma;\Delta\vdash A\quad\Gamma;\Delta\vdash B}{\Gamma;\Delta\vdash A\wedge B}\qquad\frac{\Gamma,x:\alpha;\Delta\vdash A}{\Gamma;\Delta\vdash\forall x^{\alpha}.A}\qquad\frac{\Gamma;\Delta,x:\alpha\vdash A}{\Gamma;\Delta\vdash\forall x^{\Box\alpha}.A}\qquad\frac{\Gamma;\Delta\vdash u:\langle\alpha\rangle\quad\Gamma;\Delta,m:\alpha\vdash A}{\Gamma;\Delta\vdash u=\langle m\rangle\{A\}}$$

$$\frac{\Gamma;\Delta\vdash A}{\Gamma;\Delta\vdash\neg A}\qquad\frac{\Gamma;\Delta\vdash A\quad m\notin\mathsf{dom}(\Gamma)\cup\mathsf{dom}(\Delta)\quad\Gamma;\Delta\vdash M:\alpha\quad\Gamma,m:\alpha;\Delta\vdash B}{\Gamma;\Delta;\alpha\vdash\{A\}\,M:_m\{B\}}$$

Fig. 5. Typing rules for expressions, formulae and judgements. Rules for constants and first-order operations omitted.