Static Name Control for FreshML

François Pottier
INRIA
Francois.Pottier@inria.fr

Abstract

FreshML extends ML with constructs for declaring and manipulating abstract syntax trees that involve names and statically scoped binders. It is impure: name generation is an observable side effect. In practice, this means that FreshML allows writing programs that create fresh names and unintentionally fail to bind them. Following in the steps of early work by Pitts and Gabbay, this paper defines Pure FreshML, a subset of FreshML equipped with a static proof system that guarantees purity. Pure FreshML relies on a rich binding specification language, on user-provided assertions, expressed in a logic that allows reasoning about values and about the names that they contain, and on a conservative, automatic decision procedure for this logic. It is argued that Pure FreshML can express non-trivial syntax-manipulating algorithms.

1 Introduction

FreshML [16, 15] extends ML with constructs for declaring and manipulating abstract syntax trees that involve names and statically scoped binders. FreshML aims to be a better meta-programming language than ML by allowing a programming style that closely reflects the standard, semi-informal practice of reasoning "up to α-conversion".

Unfortunately, FreshML is *impure*, in the sense that fresh name generation is an observable side effect. For instance, in FreshML, one can introduce a type of λ-terms and define a function that purports to construct "the" set of *bound* names of a λ-term [9, Figure 1]. Such a function is accepted, and produces two *distinct* sets of names if applied twice to the same term! This is undesirable: one would like to be able to define only *pure* functions, that is, functions map α-equivalent arguments to α-equivalent results.

As another facet of the same problem, a FreshML meta-program can construct terms that accidentally contain unbound names. Again, this is undesirable: one would like to be warned by the compiler when a meta-program *generates* a fresh name, but fails to eventually *bind* it.

State of the art This deficiency is a known problem. Pitts and Gabbay attacked it by equipping FreshML 2000 [9] with a static "freshness inference" system whose purpose was to rule out all impure uses of the fresh name generation facility. FreshML 2000 achieves this goal, but is too conservative. For this reason, static name control was abandoned by Shinwell, Pitts, and Gabbay in later work [16].

The problem is not specific to FreshML. To the best of my knowledge, it is shared by most meta-programming languages in existence today. One exception is MetaML [14], which avoids the problem in an interesting way. In MetaML, the idiom $\langle \mathsf{fn}\ x \Rightarrow \tilde{}e \rangle$ *generates* a fresh name, denoted by the meta-variable x, evaluates the expression e, producing an abstract syntax tree $\langle t \rangle$ that can contain free occurrences of the name denoted by x, and *binds* that name by constructing the abstract syntax tree $\langle \mathsf{fn}\ x \Rightarrow t \rangle$, which is returned. In this design, the operations of *generating* and *binding* names cannot be separated. This guarantees purity, but comes at a heavy cost in expressiveness: it is often useful, or necessary, to view these operations as separate. For a similar reason, MetaML allows the *construction* of code fragments, but not their *deconstruction*: it does not offer an analogue of FreshML's `case` construct, which inspects a piece of abstract syntax via pattern matching. In FreshML, matching against an *abstraction* pattern *generates* a fresh name, but does not *bind* it.

Cαml (pronounced: "alphaCaml") [10] can be thought of as a tool that provides much of the power of FreshML to Objective Caml users. The tool accepts so-called *binding specifications*, that is, algebraic data type declarations, enriched with information on where and how atoms are bound. The tool turns these specifications into Objective Caml type declarations and code. By relying on Objective Caml's abstract types, it is able to guarantee that atoms of different sorts are not mixed, and that abstractions (in FreshML's sense) are not violated—that is, their bound atoms are "freshened" when they are deconstructed. However, the tool contains no type system or proof system of its own, so it cannot guarantee that its fresh name generation facility is used in a pure manner.

Other pieces of related work are discussed in §6.

Towards a solution This paper presents Pure FreshML, a version of FreshML equipped with a static discipline for enforcing purity. I refer to this discipline as a *proof system*, rather than a type system, because it is very much like a Hoare logic for proving properties of programs. Throughout the paper, I use the words "pure" and "purity" in a somewhat non-standard fashion: in a "pure" program, name generation is not an observable side effect, but non-termination remains possible.

The proof system is layered on top of a conventional *type system*, which, in this paper, is a system of simple types. Enriching the type system with more features, such as ML- or System F-style polymorphism, would be straightforward. In fact, the proof system is almost entirely independent of the underlying type system. The only connection between the two resides in the interpretation of constraints (§3), which is typed: that is, the type of a variable can influence the meaning of a constraint. If desired, the type system can have type inference: the presence of the proof system does not prevent that.

The proof system is inspired by Pitts and Gabbay's "freshness inference" system [9], but is significantly more expressive, thanks to three new ingredients.

First, the system relies on a logic that combines Boolean constraints over sets of atoms, equations between values, and the primitive function *fa*, also known as *support*, which maps a value to the set of its free atoms. The judgements of the proof system involve Hoare-style triples of the form $\{H\}\, e\, \{P\}$, where H is a constraint—a precondition—and P is a parameterized constraint—a postcondition. The logic comes with a fully automated decision procedure for entailment problems, which is sound, and slightly conservative.

Second, the system allows explicit assertions to be provided by the programmer. Function definitions are annotated with optional *preconditions* and *postconditions*. Similarly, let constructs carry an optional postcondition. Last, data constructor declarations carry an optional *guard*.

Last, the system relies on Cαml's *binding specification* language [10] as a means of describing how names are bound. The need for an expressive binding specification language arises not only when dealing with complex abstract syntax, but also when defining internal data structures that involve names, such as evaluation environments (§5) and nested, name-capturing contexts [12, §6.2].

Expressiveness Due to space restrictions, this extended abstract describes a version of Pure FreshML equipped with ordinary (as opposed to generalized) algebraic data types and with FreshML's (as opposed to Cαml's) binding specification language. The missing features are described in the full version of the paper [12, §5].

In this paper, I demonstrate the expressiveness of the full language by presenting a small but non-trivial example program: *normalization by evaluation* (§5). The full paper contains another example: *conversion to A-normal form* [12, §6.2].

Normalization by evaluation is an interesting benchmark because it makes non-trivial use of names and environments. It is used by Shinwell *et al.* [16], who stress the ease with which it is expressed in FreshML. They point out that it is *not* accepted by FreshML 2000's static "freshness inference" system [9], and that even a manual proof of its correctness is "far from immediate" [16]. Up to a few changes and annotations, it *is* expressible in Pure FreshML.

Road map The paper is laid out as follows. First (§2), I introduce the syntax of Pure FreshML, its operational semantics, and a simple type system, which statically prevents most errors, but does *not* prevent incorrect uses of the name generator. Then (§3), I define the syntax and interpretation of constraints, as well as a conservative decision procedure for entailment problems. Equipped with these tools, I introduce the proof system (§4) and prove that it statically prevents *all* errors. An example is presented in §5. The paper ends with discussions of related and future work (§6, §7). All proofs, as well as many details and digressions, are omitted in this extended abstract. The interested reader is again referred to the full version of the paper [12]. An early prototype implementation, together with several code samples, is available online [11].

2 Pure FreshML

2.1 Syntax

The syntax of Pure FreshML appears in Figure 1. It is similar to the calculi of Pitts *et al.* [9, 16], up to the omission of first-class functions (see §7 for a discussion).

Values v include variables x, the unit value $()$, pairs (v, v), injections $K\, v$, where K ranges over data constructors, and binary *abstractions* $\langle x \rangle\, v$, where the variable x denotes an atom—that is, an object-level name. In an abstraction $\langle x \rangle\, v$, the variable x is not bound: this is a free occurrence of x. Patterns p are shallow and form a subset of values. They are required to be linear.

In order to facilitate the formulation of the proof system, a couple of simplifications are built into the syntax. First, the actual argument of a function call, as well as the scrutinee of a case construct, must be values. This requirement, which is reminiscent of A-normal form [3], is met by introducing let forms to name the results of intermediate computations. Second, the case construct only has one branch, guarded by a shallow pattern. Two exception forms, next and fail, together with a try construct, allow encoding general case constructs featuring an arbitrary number

Syntactic objects

$$
\begin{array}{rcl}
v & ::= & x \mid () \mid (v, v) \mid K\, v \mid \langle x \rangle\, v \\
p & ::= & \quad () \mid (x, x) \mid K\, x \mid \langle x \rangle\, x \\
e & ::= & v \\
 & \mid & \mathsf{case}\ p = v\ \mathsf{then}\ e \\
 & \mid & \mathsf{absurd} \mid \mathsf{next} \mid \mathsf{fail} \\
 & \mid & \mathsf{try}\ e\ \mathsf{else}\ e \\
 & \mid & \mathsf{fresh}\ x\ \mathsf{in}\ e \\
 & \mid & \mathsf{if}\ x = x\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & \mid & \mathsf{let}\ x\ \mathsf{where}\ C = e\ \mathsf{in}\ e \\
 & \mid & f(v) \\
fd & ::= & \mathsf{fun}\ f(x\ \mathsf{where}\ C) : x\ \mathsf{where}\ C = e \\
C, H & ::= & (\text{see } \S 3.1) \\
\tau & ::= & \mathsf{atom} \mid \mathsf{unit} \mid \tau \times \tau \mid \delta \mid \langle \mathsf{atom} \rangle\, \tau \\
\Gamma & ::= & \epsilon \mid \Gamma; x : \tau
\end{array}
$$

Semantic objects

$$
\begin{array}{rcl}
w & ::= & a \mid () \mid (w, w) \mid K\, w \mid \langle a \rangle\, w \\
F & ::= & a \mid x.e \mid x = w \mid e \\
S & ::= & \epsilon \mid S; F \\
c & ::= & S/e \mid S/w
\end{array}
$$

Mapping values to semantic values

$$
\begin{array}{rcll}
\rho(()) & = & () & \\
\rho((v_1, v_2)) & = & (\rho(v_1), \rho(v_2)) & \\
\rho(K\, v) & = & K\, \rho(v) & \\
\rho(\langle x \rangle\, v) & = & \langle \rho(x) \rangle\, \rho(v) & \text{if } \rho(x) \text{ is an atom}
\end{array}
$$

Figure 1. Syntax of Pure FreshML

of branches and deep (nested) patterns. These simplifications make Pure FreshML, as presented here, a kernel language. In practice, one would offer an unrestricted surface language and define a translation from the surface language down to the kernel language. This is done in my prototype implementation.

Expressions can build values via "v" and deconstruct them via "case $p = v$ then e", where the variables in p are considered bound within e. The execution of a case construct aborts, by raising next, if p does not match v. next is an exception that is caught with a try construct. fail is an exception that cannot be caught. absurd asserts that the current program point is unreachable. It is somewhat similar to fail, but is statically checked, so, in a valid program, it is never executed. As in Pitts and Gabbay's paper [9], the if construct is specialized: it compares two atoms for equality.

The fresh construct generates a fresh atom. As in Pitts and Gabbay's original work [9], the atom is bound to a variable x whose scope is the expression e. In contrast, in Shinwell *et al.*'s later work [16], fresh is just an effectful primitive operation. It seems that only the first form can be given a pure semantics, so it is naturally the one I adopt.

The let form is standard, except for the assertion C. In let x where $C = e_1$ in e_2, the variable x is bound in C and e_2. The constraint C acts as a postcondition for e_1, and must in general be explicitly supplied by the user. (Automatically computing a strongest postcondition for an arbitrary expression is not possible, because the constraint logic is too weak—for instance, it lacks existential quantification.) Yet, in certain common cases, the translation from the surface language down to the kernel language can make up an appropriate constraint. For instance, if e_1 is a value v, then $x = v$ is the strongest postcondition. If e_1 is a function call $f(v)$, then the postcondition associated with f, instantiated with v and x, is the strongest postcondition.

A program is composed of a set of mutually recursive toplevel function definitions. Each such definition takes the form fun $f(x_1$ where $C_1) : x_2$ where $C_2 = e$, where x_1 is bound within C_1, C_2, and e, while x_2 is bound only within C_2. This defines a function whose precondition is C_1 and whose postcondition is C_2.

2.2 Operational semantics

I have pointed out that, in an abstraction $\langle x \rangle\, v$, the variable x is not bound in v. Yet, an intuitive understanding of the semantics of FreshML dictates that, when this abstraction is evaluated, the atom denoted by x *becomes bound* in the value denoted by v. In order to formalize this intuition, I introduce a distinct syntactic category of *semantic values*, written w (Figure 1).

Semantic values do not contain variables, but contain *atoms* a, drawn from a countably infinite set $\mathbb{A}$, and contain abstractions of the form $\langle a \rangle\, w$, where a *is considered bound in* w. The set of *free atoms* of a semantic value w, written $fa(w)$, is defined in the obvious way. It is also known as the *support* of w. An atom a is *fresh for* some syntactic entity when it is not among the free atoms of that entity.

Values v contain variables, but not atoms, while semantic values w contain atoms, but not variables. Values are turned into semantic values via *simultaneous* substitution of semantic values for *all* free variables. In order to maintain a strict segregation between values and semantic values, the operational semantics relies on *stacks*, which, among other roles, represent a deferred substitution of semantic values for all variables in scope.

A *stack* S is a sequence of *frames* F (Figure 1). The presence of the frame a on the stack means that a fresh construct was entered, that the freshly generated atom is a, and that the fresh construct was not exited yet. The frame $x.e$ means that the left-hand side of a let construct was entered. The value of the left-hand side, when available, will be bound

$$
\begin{array}{rll}
S/v & \longrightarrow S/S(v) & & (1)\\
S/\mathsf{case}\ () = v\ \mathsf{then}\ e & \longrightarrow S/e & \text{if } S(v) = () & (2)\\
S/\mathsf{case}\ (x_1, x_2) = v\ \mathsf{then}\ e & \longrightarrow S; x_1 = w_1; x_2 = w_2/e & \text{if } S(v) = (w_1, w_2) & (3)\\
S/\mathsf{case}\ K_1\, x = v\ \mathsf{then}\ e & \longrightarrow S; x = w/e & \text{if } S(v) = K_2\, w \text{ and } K_1 = K_2 & (4)\\
S/\mathsf{case}\ K_1\, x = v\ \mathsf{then}\ e & \longrightarrow S/\mathsf{next} & \text{if } S(v) = K_2\, w \text{ and } K_1 \neq K_2 & (5)\\
S/\mathsf{case}\ \langle x_1\rangle\, x_2 = v\ \mathsf{then}\ e & \longrightarrow S; a; x_1 = a; x_2 = w/e & \text{if } S(v) = \langle a\rangle\, w \text{ and } a \text{ fresh for } S & (6)\\
S/\mathsf{try}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow S; e_2/e_1 & & (7)\\
S; e/\mathsf{next} & \longrightarrow S/e & & (8)\\
S; F/\mathsf{next} & \longrightarrow S/\mathsf{next} & \text{except if the previous rule applies} & (9)\\
S; F/\mathsf{fail} & \longrightarrow S/\mathsf{fail} & & (10)\\
S/\mathsf{fresh}\ x\ \mathsf{in}\ e & \longrightarrow S; a; x = a/e & \text{if } a \text{ fresh for } S & (11)\\
S/\mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow S/e_1 & \text{if } S(x_1) = a_1 \text{ and } S(x_2) = a_2 \text{ and } a_1 = a_2 & (12)\\
S/\mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow S/e_2 & \text{if } S(x_1) = a_1 \text{ and } S(x_2) = a_2 \text{ and } a_1 \neq a_2 & (13)\\
S/\mathsf{let}\ x\ \mathsf{where}\ C = e_1\ \mathsf{in}\ e_2 & \longrightarrow S; x.e_2/e_1 & & (14)\\
S/f(v) & \longrightarrow S; x_1 = S(v)/e & \text{if } \mathsf{fun}\ f(x_1)\ \ldots = e & (15)\\
S; a/w & \longrightarrow S/w & \text{if } a \text{ fresh for } w & (16)\\
S; x.e/w & \longrightarrow S; x = w/e & & (17)\\
S; x = w'/w & \longrightarrow S/w & & (18)\\
S; e/w & \longrightarrow S/w & & (19)
\end{array}
$$

Figure 2. Operational semantics

to x in the evaluation of e. Note that x is considered bound within e. The frame $x = w$ means that x is currently bound to the semantic value w. The frame e means that a try construct was entered, and was not exited yet. If the exception next is raised, it will be caught and e will be evaluated; if, on the other hand, a value is returned, e will be discarded.

I define the *domain* of a frame F as follows. The domain of a is a; the domain of $x = w$ is x; the domain of $x.e$ and of e is empty. The domain of a stack S is the ordered sequence of variables and atoms obtained by concatenating the domains of the frames that make up S. A syntactic entity is *closed under* S when its free variables and free atoms are members of the domain of S.

A *configuration* is of the form S/e or S/w, where e and w are closed under S. The variables and atoms in the domain of S are considered bound in such a configuration, so that configurations are closed. A *result* is a configuration of the form ϵ/w or ϵ/next or ϵ/fail.

A *valuation* ρ is a finite mapping of variables to semantic values. It is lifted to a mapping of values to semantic values (Figure 1). Note that a syntactic abstraction $\langle x\rangle\, v$ is mapped down to a semantic abstraction $\langle \rho(x)\rangle\, \rho(v)$, where the atom $\rho(x)$ is now bound in the semantic value $\rho(v)$. If $\rho(x)$ happens not to be an atom, then $\rho(\langle x\rangle\, v)$ is undefined. Such a situation is ruled out by the type system (§2.3).

A stack S can be viewed as a valuation, defined by the collection of all frames of the form $x = w$ within S. Thus, a value v that is closed under a stack S can be turned into a semantic value $S(v)$.

The small-step operational semantics of Pure FreshML is given by a binary reduction relation over configurations (Figure 2). The rules may seem numerous, but are simple. I now explain some of them.

Reduction rule 1 turns a value v into a semantic value; it is applicable only if $S(v)$ is defined. Reduction rules 2–6, 12–13, and 15 also exploit this mechanism.

Reduction rule 11 states that evaluating "fresh x in e" creates a fresh atom a, augments the stack with two new frames, which separately record the fact that a was created and the fact that x was bound to a, and proceeds with the evaluation of e. When and if e eventually reduces to a semantic value w, these two stack frames are popped by reduction rules 18 and 16, *provided a does not appear free in w*. This requirement is directly inspired by Gabbay and Pitts' treatment of "locally fresh atoms" [4, Remark 6.4]. When the side condition of reduction rule 16 is violated, no reduction is possible: the configuration $S; a/w$ is stuck. This corresponds to an incorrect use of the fresh construct,

which one would like to statically prevent.

Reduction rules 2–6 describe pattern matching. In particular, reduction rule 5 states that the failure of pattern matching causes the exception next to be raised. Reduction rule 6 states that matching against an abstraction pattern $\langle x_1 \rangle\, x_2$ causes a fresh atom a to be generated, just as if a fresh construct had been evaluated [9, 16, 15].

The semantics is pure, in the sense that it does not rely on global state, as would be necessary if the creation of fresh atoms was an uncontrolled side effect [16]. Here, the stack discipline ensures that the dynamic extent of a fresh atom does not exceed the static scope of the fresh construct. According to this semantics, a program that attempts to exploit fresh in an impure manner goes wrong: it reduces to a stuck configuration. Thus, the slogan "valid programs cannot go wrong", which I establish later (Theorem 4.3), means that valid programs are in fact pure.

2.3 Type system

I equip Pure FreshML with a conventional system of simple types [16]. The proof system relies on it in only two ways: to guarantee that only well-formed values appear in constraints, and to obtain information about the support of a variable, based on its type (§3.3).

The types (Figure 1) are Pitts' nominal arities [8, §2.2]. Every data constructor K carries a signature of the form $\tau \to \delta$, where δ is a data type. Every function f carries a signature of the form $\tau_1 \to \tau_2$. The definition of the type system appears in the full version of this paper [12].

Theorem 2.1 (Subject reduction) *A well-typed configuration can reduce only to a well-typed configuration.* ◇

Theorem 2.2 (Partial Progress) *A well-typed, irreducible configuration is either a result, or of the form* S/absurd, *or of the form* $S; a/w$, *where* a *occurs free in* w. ◇

The statement of Theorem 2.2 pinpoints the basic issue that this paper addresses: a conventional type system does not guarantee that a Pure FreshML program cannot go wrong. A well-typed Pure FreshML program *can* go wrong, either by attempting to execute an absurd statement, or by letting a fresh-bound atom escape its static scope.

3 Constraints

I now present the constraint logic and the decision procedure for entailment problems that underlie Pure FreshML's proof system. This is done in several steps. I first introduce the syntax and interpretation of constraints (§3.1). Then, I present a sound, conservative decision procedure for entailment problems. It is defined via a reduction to SAT, in three steps: elimination of all value equations (§3.2), elimination of all applications of *fa* (§3.3), and switch from the Boolean algebra $\mathcal{P}(\mathbb{A})$ to the Boolean algebra $\mathbb{B}$ (§3.4). The decision procedure is sound, but incomplete. There are two sources of incompleteness, discussed in §3.2 and §3.3.

3.1 Syntax and interpretation

Here is the syntax of *set expressions* s and *constraints* C:

$$\begin{array}{rcl} s & ::= & \mathit{fa}(v) \mid \emptyset \mid \mathbb{A} \mid s \cap s \mid s \cup s \mid \neg s \\ C, H & ::= & s = \emptyset \mid s \neq \emptyset \mid v = v \mid C \wedge C \end{array}$$

The set expression $s_1 \setminus s_2$, as well as the constraints *false*, *true*, $s_1 \subseteq s_2$, $s_1 = s_2$, and $s_1 \# s_2$ can be viewed as sugar. The last of these stands for $(s_1 \cap s_2) = \emptyset$. Set expressions denote sets of atoms, that is, elements of the Boolean algebra $\mathcal{P}(\mathbb{A})$. Constraints are conjunctions of *atomic constraints*: set emptiness (or non-emptiness) assertions and value equations.

Constraints are typed. For a constraint C to be well-typed under environment Γ, (i) if a value v appears within C, then v must be well-typed under Γ, and (ii) if a value equation $v_1 = v_2$ appears within C, then v_1 and v_2 must have the same type under Γ. Throughout the paper, I manipulate constraints without explicitly mentioning under which type environment Γ they are to be considered.

A valuation ρ *respects* a type environment Γ if it maps every variable x in the domain of Γ to a semantic value of type $\Gamma(x)$. The satisfaction judgement $\rho \vdash C$ is defined when C is well-typed under Γ and ρ respects Γ. I omit its formal definition. In short, the interpretation of a value v under ρ is $\rho(v)$ (Figure 1). The symbol *fa* maps the semantic values into $\mathcal{P}(\mathbb{A})$. Value equations are interpreted in terms of equality of semantic values (which, at atom abstractions, involves α-equivalence). *Satisfiability* and *entailment* are defined in the standard way. I write $\vdash C$ when C is satisfiable and $C_1 \Vdash C_2$ when C_1 entails C_2.

3.2 Eliminating value equations

I assume that the right-hand side of every entailment problem is a set constraint (as opposed to a value equation). That is, a value equation can only be a hypothesis, not a goal. All problems emitted by the proof system in §4 satisfy this assumption. Then, entailment problems are easily reduced to satisfiability problems.

I now explain how to reduce an arbitrary constraint C to a constraint C' that contains no value equations, in such a way that, if C is satisfiable, then so is C'. This transformation is *sound* in the sense that, modulo the reduction of entailment down to satisfiability, it leads to a conservative decision procedure for entailment problems.

The idea is simple: first, examine the value equations in C and discover as many of their consequences as possible,

including new value equations and new set constraints; then, drop all value equations.

The first step can be viewed as a closure computation, defined by the following rules:

$$\begin{array}{rl} v_1 = v_2 \rightarrow v_2 = v_1 & \\ v_1 = v_2 \wedge v_2 = v_3 \rightarrow v_1 = v_3 & \\ (v_1, v_1') = (v_2, v_2') \rightarrow v_1 = v_2 \wedge v_1' = v_2' & \\ K\, v_1 = K\, v_2 \rightarrow v_1 = v_2 & \\ K_1\, v_1 = K_2\, v_2 \rightarrow \mathit{false} & \text{if } K_1 \neq K_2 \\ v_1 = v_2 \rightarrow \mathit{fa}(v_1) = \mathit{fa}(v_2) & \end{array}$$

It is clear that each of the rules preserves the interpretation of the constraint, so this step is sound and complete. One rule that I have purposely omitted, because it is not interpretation-preserving, is the following:

$$\langle x_1 \rangle\, v_1 = \langle x_2 \rangle\, v_2 \rightarrow x_1 = x_2 \wedge v_1 = v_2 \qquad \text{(unsound)}$$

This rule is incorrect, because equality of abstractions is not syntactic—that is the whole point of abstractions!

The second step consists in dropping all value equations. It is clearly sound. It is also incomplete, because of the missing closure rule for abstractions.

3.3 Eliminating applications of *fa*

I now explain how to reduce a constraint C (without value equations) to a Boolean constraint C' in such a way that, if C is satisfiable, then C' is satisfiable as well, when interpreted over $\mathcal{P}(\mathbb{A})$.

The syntax of *Boolean constraints* is as follows:

$$\begin{array}{rcl} s & ::= & X \mid 0 \mid 1 \mid s \wedge s \mid s \vee s \mid \neg s \\ C & ::= & s = 0 \mid s \neq 0 \mid C \wedge C \end{array}$$

Here, X ranges over a new category of *Boolean variables*. Boolean constraints can be interpreted over any Boolean algebra. In particular, when they are interpreted over $\mathcal{P}(\mathbb{A})$, a Boolean variable X denotes a set of atoms. (In that case, I also refer to X as a *set variable*.) When they are interpreted over the two-point algebra $\mathbb{B} = \{0, 1\}$, such a variable denotes a truth value.

An atomic constraint of the form $s = 0$ is *positive*; an atomic constraint of the form $s \neq 0$ is *negative*. A conjunction of atomic constraints that contains at most one negative conjunct is *simple*.

In order to perform the reduction announced above, only one transformation is required: to replace all applications of the *fa* symbol with set variables. This is done is two steps.

First, applications of *fa* are reduced:

$$\begin{array}{rl} \mathit{fa}(()) & \rightarrow \emptyset \\ \mathit{fa}((v_1, v_2)) & \rightarrow \mathit{fa}(v_1) \cup \mathit{fa}(v_2) \\ \mathit{fa}(K\, v) & \rightarrow \mathit{fa}(v) \\ \mathit{fa}(\langle x \rangle\, v) & \rightarrow \mathit{fa}(v) \setminus \mathit{fa}(x) \end{array}$$

As a result, only applications of the form $\mathit{fa}(x)$ remain.

Second, each occurrence of $\mathit{fa}(x)$, where x has type τ, is rewritten as follows:

1. if every semantic value of type τ has empty support, then $\mathit{fa}(x)$ is replaced with $\emptyset$;

2. if no semantic value of type τ has empty support, then $\mathit{fa}(x)$ is replaced with a set variable X, and the conjunct $X \neq \emptyset$ is added to the constraint;

3. otherwise, $\mathit{fa}(x)$ is replaced with a set variable X.

The basic idea behind this transformation resides in the third rule: $\mathit{fa}(x)$ is considered an unknown set of atoms, so a set variable, written X, is introduced to stand for it. (I assume a one-to-one correspondence between variables x and set variables X.) The result is a Boolean constraint. Rules 1 and 2 are not required for the transformation to be sound. Instead, they help bring it "closer to completeness". I now discuss each of these two rules, as well as the issue of incompleteness, in turn.

Rule 1 states that, if x has type τ and if every value of type τ has empty support, then $\mathit{fa}(x)$ must be empty. (Pitts and Gabbay [9] refer to such a type τ as "pure".) This is the case if τ is a base type, such as bool, int, or string. It is also the case if τ is a data type δ and if one can prove, by structural induction, that all values of type δ have empty support. Such a proof is easily automated, so that it is decidable whether rule 1 is applicable.

Rule 2 is, in a way, the dual of rule 1. It is applicable, for instance, if τ is atom, or a data type of non-empty lists of atoms. In that case, $\mathit{fa}(x)$ is replaced with a set variable X, as in rule 3, but, in addition, the hypothesis $X \neq \emptyset$ is introduced.

This rule is important because it is the only source of negative hypotheses in the entire system. If it was removed, then all of the entailment problems produced by the proof system would carry positive hypotheses only. Why would that be a problem? Notice that the positive Boolean constraints that the system produces are somewhat peculiar. Because they exploit the connectives $\emptyset$, $\cup$, $\setminus$, but do not exploit the connectives $\mathbb{A}$ and $\neg$, they are always satisfied by the valuation that maps every Boolean variable to $\emptyset$. This means that, in the absence of rule 2, the current set of hypotheses H would always be satisfiable. So, the entailment assertion $H \Vdash \mathit{false}$ would never hold, and the expression absurd would never be accepted by the proof system (see rule ABSURD in Figure 3). In short, negative hypotheses of the form $X \neq \emptyset$ are required in order to establish absurdity.

The transformation performed in the second step is not complete: it can turn an unsatisfiable constraint into a satisfiable Boolean constraint. For instance, if x, x_1, and x_2

have type atom, then the constraint

$$\begin{array}{l} \mathit{fa}(x_1) \# \mathit{fa}(x_2) \wedge \\ \mathit{fa}(x_1) \cup \mathit{fa}(x_2) \subseteq \mathit{fa}(x) \end{array}$$

is unsatisfiable, because it requires $\mathit{fa}(x)$ to have cardinal 2, which is impossible—the support of an atom is a singleton. Yet, it is reduced to the Boolean constraint

$$\begin{array}{l} X_1 \cap X_2 = \emptyset \wedge \\ X_1 \cup X_2 \subseteq X \wedge \\ X \neq \emptyset \wedge X_1 \neq \emptyset \wedge X_2 \neq \emptyset \end{array}$$

which is satisfiable over $\mathcal{P}(\mathbb{A})$—take $X_1 = \{a_1\}$, $X_2 = \{a_2\}$, and $X = \{a_1, a_2\}$, where a_1 and a_2 are distinct atoms. In summary, the decision procedure does distinguish between empty and non-empty sets of atoms, but is unable to reason about cardinality.

3.4 Satisfiability of Boolean constraints

I now focus on the satisfiability problem for Boolean constraints (as defined in §3.3) interpreted over the Boolean algebra $\mathcal{P}(\mathbb{A})$.

Marriott and Odersky [5] have shown that any Boolean algebra of infinite height is weakly independent. This means that satisfiability of arbitrary constraints reduces to satisfiability of simple constraints:

Lemma 3.1 *Let C be a conjunction of positive atomic constraints. The constraint $C \wedge s_1 \neq 0 \wedge \ldots \wedge s_n \neq 0$, where $n > 0$, is satisfiable over $\mathcal{P}(\mathbb{A})$ if and only if each of the simple constraints $C \wedge s_i \neq 0$ is satisfiable over $\mathcal{P}(\mathbb{A})$.* ◇

There remains to explain how to decide whether a simple constraint is satisfiable. I establish the following result:

Lemma 3.2 *A simple constraint is satisfiable over $\mathcal{P}(\mathbb{A})$ if and only if it is satisfiable over $\mathbb{B}$.* ◇

When interpreted over $\mathbb{B}$, the atomic constraint $s \neq 0$ is equivalent to $(\neg s) = 0$. As a result, determining whether a constraint is satisfiable over $\mathbb{B}$ is exactly the Boolean satisfiability problem SAT.

4 A proof system

I now define the proof system that lies at the heart of Pure FreshML. It can be viewed as an algorithm that extracts proof obligations out of a Pure FreshML program. Each proof obligation is an entailment problem and is discharged using the decision procedure of §3. As explained there, the decision procedure needs access to type information. However, the proof system *per se* does not, so I do not keep track of types in this section.

4.1 Presentation

The proof system consists of three main judgements, which concern patterns, expressions, and function definitions (Figure 3).

Judgements about expressions are of the form $\Delta \vdash \{H\}\, e\, \{P\}$. Δ is a set of all variables currently in scope, and includes the free variables of e. I implicitly assume that e is well-typed under a type environment whose domain is Δ. H is a constraint. It represents a precondition, that is, a hypothesis. (I use C and H for constraints.) P is a predicate: a constraint, parameterized over one variable. It represents a postcondition, that is, a goal.

I sometimes explicitly write $\lambda x.C$ for a parameterized constraint: then, the parameter x stands for the result of the expression e. When P is $\lambda x.C$, I write $P(v)$ for $[x \mapsto v]C$, where v is a value. I write $C[\cdot]$ for the predicate $\lambda x.C[x]$, where x is chosen fresh for C. I write *true* for the predicate $\lambda x.\mathit{true}$. I write $P_1 \wedge P_2$ for the predicate $\lambda x.(P_1(x) \wedge P_2(x))$, where x is fresh for P_1 and P_2.

Rule VALUE states that the triple $\{H\}\, v\, \{P\}$ is satisfied if and only if the precondition H entails that the value v satisfies the postcondition P. Its premise, an entailment judgement, represents a proof obligation.

Rule FRESH augments H with the hypothesis $\mathit{fa}(x) \# \mathit{fa}(\Delta)$. (I write $\mathit{fa}(\Delta)$ for the symbolic union of all $\mathit{fa}(y)$, where y ranges over Δ.) This means that the support of x can safely be assumed disjoint with the support of every pre-existing variable. FRESH also augments the postcondition with the new goal $\mathit{fa}(x) \# \mathit{fa}(\cdot)$, that is, the atom x should not appear in the support of the result that is eventually produced by the fresh construct. This goal clearly reflects the side condition of reduction rule 16.

Rule CASE describes what can be assumed, and what must be proved, when a value v is successfully matched against a pattern p. First, the equation $p = v$ can be assumed. Second, an extra hypothesis H' and an extra goal P' are derived from the pattern p, using either ABSTRACTION-PATTERN or OTHER-PATTERN. When p is an abstraction pattern $\langle x_1 \rangle\, x_2$, H' states that x_1 can be assumed to be fresh and P' states that x_1 must not appear in the result of evaluating e, just as if x_1 was fresh-bound. When p is another pattern form, H' and P' are empty.

Rule IF augments H, in each branch, with a constraint that reflects the outcome of the dynamic test. Because x_1 and x_2 have type atom, $\mathit{fa}(x_1) \# \mathit{fa}(x_2)$ is equivalent to, and can be used instead of, $\mathit{fa}(x_1) \neq \mathit{fa}(x_2)$, a disequation that the constraint language is not directly able to express.

Rule LETWHERE uses $\lambda x.C$, where C is supplied by the user, as a postcondition for e_1, and makes $C \wedge \mathit{fa}(x) \subseteq \mathit{fa}(\Delta)$ a new hypothesis for the continuation e_2.

Rule DEF states that the body of a function must be checked under the precondition C_1 and postcondition C_2

$$\textsc{Abstraction-Pattern}\quad \Delta \vdash \{\mathit{fa}(x_1) \# \mathit{fa}(\Delta)\}\ \langle x_1\rangle\ x_2\ \{\mathit{fa}(x_1) \# \mathit{fa}(\cdot)\}$$

$$\textsc{Other-Pattern}\quad \frac{p \not\equiv \langle x_1\rangle\ x_2}{\Delta \vdash \{\mathit{true}\}\ p\ \{\mathit{true}\}} \qquad \textsc{Value}\quad \frac{H \Vdash P(v)}{\Delta \vdash \{H\}\ v\ \{P\}}$$

$$\textsc{Case}\quad \frac{\mathrm{dom}(p)\text{ fresh for }\Delta, H, v, P \qquad \Delta \vdash \{H'\}\ p\ \{P'\} \qquad \Delta, \mathrm{dom}(p) \vdash \{H \wedge H' \wedge p = v\}\ e\ \{P \wedge P'\}}{\Delta \vdash \{H\}\ \mathsf{case}\ p = v\ \mathsf{then}\ e\ \{P\}}$$

$$\textsc{Absurd}\quad \frac{H \Vdash \mathit{false}}{\Delta \vdash \{H\}\ \mathsf{absurd}\ \{P\}} \qquad \textsc{Next}\quad \Delta \vdash \{H\}\ \mathsf{next}\ \{P\}$$

$$\textsc{Fail}\quad \Delta \vdash \{H\}\ \mathsf{fail}\ \{P\} \qquad \textsc{Try}\quad \frac{\Delta \vdash \{H\}\ e_1\ \{P\} \qquad \Delta \vdash \{H\}\ e_2\ \{P\}}{\Delta \vdash \{H\}\ \mathsf{try}\ e_1\ \mathsf{else}\ e_2\ \{P\}}$$

$$\textsc{Fresh}\quad \frac{x\text{ fresh for }\Delta, H, P \qquad \Delta, x \vdash \{H \wedge \mathit{fa}(x) \# \mathit{fa}(\Delta)\}\ e\ \{P \wedge \mathit{fa}(x) \# \mathit{fa}(\cdot)\}}{\Delta \vdash \{H\}\ \mathsf{fresh}\ x\ \mathsf{in}\ e\ \{P\}}$$

$$\textsc{If}\quad \frac{\Delta \vdash \{H \wedge \mathit{fa}(x_1) = \mathit{fa}(x_2)\}\ e_1\ \{P\} \qquad \Delta \vdash \{H \wedge \mathit{fa}(x_1) \# \mathit{fa}(x_2)\}\ e_2\ \{P\}}{\Delta \vdash \{H\}\ \mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \{P\}}$$

$$\textsc{LetWhere}\quad \frac{x\text{ fresh for }\Delta, H, P \qquad \Delta \vdash \{H\}\ e_1\ \{\lambda x.C\} \qquad \Delta, x \vdash \{H \wedge C \wedge \mathit{fa}(x) \subseteq \mathit{fa}(\Delta)\}\ e_2\ \{P\}}{\Delta \vdash \{H\}\ \mathsf{let}\ x\ \mathsf{where}\ C = e_1\ \mathsf{in}\ e_2\ \{P\}}$$

$$\textsc{Call}\quad \frac{H \Vdash \mathrm{pre}(f)(v) \qquad H \Vdash \mathrm{post}(f)(v, \cdot) \Rightarrow P}{\Delta \vdash \{H\}\ f(v)\ \{P\}} \qquad \textsc{Def}\quad \frac{x_1 \vdash \{C_1\}\ e\ \{\lambda x_2.C_2\} \qquad \mathrm{pre}(f) = \lambda x_1.C_1 \qquad \mathrm{post}(f) = \lambda(x_1, x_2).(C_2 \wedge \mathit{fa}(x_2) \subseteq \mathit{fa}(x_1))}{\vdash \mathsf{fun}\ f(x_1\ \mathsf{where}\ C_1) : x_2\ \mathsf{where}\ C_2 = e}$$

Figure 3. The proof system

that were provided by the user. It also defines the notations "$\mathrm{pre}(f)$" and "$\mathrm{post}(f)$" used in rule CALL.

4.2 Soundness

The combination of the type system and proof system is sound with respect to the operational semantics. This is proven via standard subject reduction and progress results. A configuration is *valid* when it is accepted by the type system and proof system.

Theorem 4.1 (Subject Reduction) *A valid configuration can reduce only to a valid configuration.* ◇

Theorem 4.2 (Progress) *A valid, irreducible configuration is a result.* ◇

Corollary 4.3 (Soundness) *A valid configuration cannot go wrong.* ◇

5 Illustration

I now briefly present a small but non-trivial program that is accepted by the proof system: normalization by evaluation (Figure 4). It is adapted from Shinwell *et al.* [16]. The three main changes are: the code is defunctionalized; the binding structure of closures and environments is expressed using Cαml's specification language (not formally presented in this paper, see [10] and [12, §5]); and *evals* carries an explicit postcondition.

The effect of defunctionalization is that the data constructor L (line 7) carries a triple of an environment, an atom, and a term, instead of a first-class function.

The type *env* is a "pattern type", in Cαml parlance; this is indicated by the ***binds*** keyword on line 14. This means that, in a value of type *env*, such as:

$$\mathit{env} = \mathit{ECons}(\ldots \mathit{ECons}(\mathit{ENil}, x_1, v_1) \ldots, x_n, v_n),$$

the atoms $x_1, \ldots, x_n$ are considered as *binding occurrences*. Then, by definition, the set expression $\mathit{bound}(\mathit{env})$ denotes the set of atoms $\{x_1, \ldots, x_n\}$. According to the ***outer*** keyword in the definition of *ECons* (line 16), the values $v_1, \ldots, v_n$ lie *outside* the scope of these atoms. Then, by definition, the set expression $\mathit{outer}(\mathit{env})$ denotes the set of atoms $\mathit{fa}(v_1) \cup \ldots \cup \mathit{fa}(v_n)$.

According to the placement of the abstraction brackets and of the ***inner*** keyword in the definition of L (line 7), within a value of the form:

$$v = L(\mathit{env}, x, t),$$

```
1  type lam =
2    | Var of atom
3    | Lam of ⟨ atom × inner lam ⟩
4    | App of lam × lam
5
6  type sem =
7    | L of ⟨ env × atom × inner lam ⟩
8    | N of neu
9
10 type neu =
11   | V of atom
12   | A of neu × sem
13
14 type env binds =
15   | ENil
16   | ECons of env × atom × outer sem
17
18 fun reify accepts s produces t =
19   case s of
20   | L (env, y, t) →
21       fresh x in
22       Lam (x, reify (evals (ECons (env, y, N (V (x))), t)))
23   | N (n) →
24       reifyn (n)
25   end
26
27 fun reifyn accepts n produces t =
28   case n of
29   | V (x) →
30       Var (x)
31   | A (n, d) →
32       App (reifyn (n), reify (d))
33   end
34
35 fun evals accepts env, t produces v
36 where free(v) ⊆ outer(env) ∪ (free(t) \ bound(env)) =
37   case t of
38   | Var (x) →
39       case env of
40       | ENil →
41           N (V (x))
42       | ECons (tail, y, v) →
43           if x = y then v
44           else evals (tail, t) end
45       end
46   | Lam (x, t) →
47       L (env, x, t)
48   | App (t1, t2) →
49       case evals (env, t1) of
50       | L (cenv, x, t) →
51           evals (ECons (cenv, x, evals (env, t2)), t)
52       | N (n) →
53           N (A (n, evals (env, t2)))
54       end
55   end
56
57 fun eval accepts t produces s =
58   evals (ENil, t)
59
60 fun normalize accepts t produces u =
61   reify (eval (t))
```

Figure 4. A sample program: NBE

the atoms in $bound(env)$, as well as the atom x, are considered bound within the term t. That is, by definition:

$$fa(v) = outer(env) \cup \big(fa(t) \setminus (bound(env) \cup \{x\})\big).$$

This explains why the proof obligation associated with the deconstruction of *Lam* on line 46 succeeds. Deconstructing *Lam* yields a "fresh" atom x, which one must prove does not appear in the support of the right-hand side $L(env, x, t)$. The fact that x is "fresh" means, in particular, that x is not in the support of *env*, which by definition includes $outer(env)$. By exploiting the above displayed equation, one finds that x is not in the support of $L(env, x, t)$, as desired. This fact is proved automatically by the conservative decision procedure of §3.

The proof obligation on line 46 corresponds, in part, to the obligation that FreshML 2000 was not able to automatically discharge [16, figure 7, lines 26–27]. By declaring that the data constructor L carries an abstraction, I have been able to get away with the deconstruction of *Lam*. Of course, as a result, new proof obligations appear wherever L is deconstructed (lines 20 and 50). Both require exploiting a non-trivial property of *evals*, which is expressed as a postcondition.

The function *evals* expects a pair of an environment *env* and a term *t*, and evaluates *t* within the context of *env*. As one might expect, any atom that appears in the support of *t* as well as in the domain of *env* is substituted out, which means that, if *evals* produces a result *v*, then (line 36):

$$fa(v) \subseteq outer(env) \cup (fa(t) \setminus bound(env))$$

This property is not automatically inferred by the system—it is a loop invariant—so it has to be explicitly provided. Then, it is easily checked.

6 Related work

This paper was inspired by Pitts and Gabbay's work on static "freshness inference" for FreshML [9]. Pitts and Gabbay's algorithm attempts to *infer* freshness assertions about values and expressions, or, equivalently, to infer an approximation of the support of values and expressions. The proof system presented in this paper is oriented purely towards *checking*. For this reason, explicit assertions must sometimes be provided at let constructs.

Pašalić and Linger [7] exploit the programming language Ωmega to define a data type that represents the abstract syntax of an object language, expressed in de Bruijn notation. The data type is parameterized in a way that guarantees that out-of-range de Bruijn indices cannot be constructed. The syntax of the object language includes non-trivial binding structures (patterns). Donnelly and Xi [2] explore a similar approach in the programming language ATS.

Schürmann *et al.*'s ∇-calculus [13] is a core meta-programming language where object-level terms are encoded using higher-order abstract syntax. There are no object-level names: both object-level and meta-level abstractions bind meta-variables. Object-level substitution is application of object-level abstractions. A type system guarantees that meta-variables cannot escape their scope. It is quite different from the proof system presented in this

paper. The construct $\nu x.e$ introduces a new meta-variable x and at the same time requires the result of evaluating e to depend only on meta-variables that were bound prior to x. This requirement is encoded via stacks of typing contexts and via a "box" type constructor that prevents exploiting the topmost context.

MetaOCaml relies on environment classifiers [17] to tell which code fragments are closed. An environment classifier is a type variable that abstracts a set of names. The code type constructor is parameterized with an environment classifier. This allows the type system to keep track, in a conservative way, of which names appear free in a code fragment. Closed code fragments are recognized by the fact that they are polymorphic in their environment classifier.

Nanevski's calculus $\nu^{\Box}$ [6] is inspired by FreshML, and, like Pure FreshML, provides a static discipline for enforcing purity. This is done by explicitly keeping track of the support of every value, and exploiting this information to ensure that freshly-created names do not escape. An important difference between $\nu^{\Box}$ and Pure FreshML is that $\nu^{\Box}$ lets the *type system* carry the support information, by parameterizing the "code" type constructor with a set of names, while Pure FreshML relies on a separate *proof system* and requires no changes to the type system. I believe that the latter approach is lighter (for instance, Nanevski's "support polymorphism" comes for free here) and potentially more expressive, because constraints can express properties other than approximations of the support of certain values. Another design difference is that $\nu^{\Box}$ is a *homogeneous, multi-level* staged programming language, while FreshML is a *heterogeneous* meta-programming language. This means, for instance, that Nanevski does not distinguish between meta-level λ-abstraction and object-level name abstraction.

7 Future work

Many features must be added in order to turn Pure FreshML into a realistic meta-programming language. Here are a few; see the full paper [12] for more:

First-class functions. I am confident that first-class functions can be introduced without difficulty. This requires extending the grammar of types with function types, carrying a precondition and a postcondition. Furthermore, Pitts and Gabbay [9] remarked that the support of a function is a subset of the combined support of its free variables. This approximation can be exploited to conservatively eliminate applications of *fa* to λ-abstractions.

Mutable state. Shared, modifiable references offer new ways for atoms to escape their scope. Calcagno *et al.* [1] attack the problem in the setting of MetaML and offer a solution that requires references to contain *closed* code fragments. An analogous restriction—to require references to contain values of *empty support*—would be easy to enforce in Pure FreshML, via proof obligations.

References

[1] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3):545–571, May 2003.

[2] K. Donnelly and H. Xi. Combining higher-order abstract syntax with first-order abstract syntax in ATS. In *ACM Workshop on Mechanized Reasoning about Languages with Variable Binding*, pages 58–63, 2005.

[3] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247, 1993.

[4] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2002.

[5] K. Marriott and M. Odersky. Negative Boolean constraints. Technical Report 94/203, Monash University, Aug. 1994.

[6] A. Nanevski. Meta-programming with names and necessity. Technical Report CMU-CS-02-123R, School of Computer Science, Carnegie Mellon University, Nov. 2002.

[7] Pašalić and N. Linger. Meta-programming with typed object-language representations. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 136–167, Oct. 2004.

[8] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.

[9] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer Verlag, 2000.

[10] F. Pottier. An overview of Cαml. In *ACM Workshop on ML*, volume 148(2) of *Electronic Notes in Theoretical Computer Science*, pages 27–52, Mar. 2006.

[11] F. Pottier. Prototype implementation of Pure FreshML, Jan. 2007.

[12] F. Pottier. Static name control for FreshML, full version, Jan. 2007.

[13] C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-calculus: Functional programming with higher-order encodings. Technical Report YALEU/DCS/TR-1272, Yale University, Nov. 2004.

[14] T. Sheard. Using MetaML: A staged programming language. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 207–239. Springer Verlag, Sept. 1998.

[15] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[16] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ACM International Conference on Functional Programming (ICFP)*, pages 263–274, Aug. 2003.

[17] W. Taha and M. F. Nielsen. Environment classifiers. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 26–37, Jan. 2003.