Static Name Control for FreshML

François Pottier
INRIA
Francois.Pottier@inria.fr

April 17, 2007

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 [23, 22] 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 [15, 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 [15] 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 [23].

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 [20], 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") [16] 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 §7.

Even though neither FreshML, MetaML, Cαml, or any deployed meta-programming language that I know of, solves this problem, it is worth attacking. It is easy, and important, to statically detect that a program *is* lexically ill-formed. It should be just as important to statically detect that a meta-program can *generate* a lexically ill-formed program.

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 [15], 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 [16] as a means of describing how names are bound. This language is more expressive than Urban, Pitts, and Gabbay's nominal signatures [25], which only allow binding one name at a time, and than Fresh Objective Caml's abstraction types [21], which do not allow binding occurrences and free occurrences to coexist within a single data structure, such as an environment. 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 (§6.1) and nested, name-capturing contexts (§6.2). It should be noted that the choice of Cαml's specification language, as opposed to some (as of now hypothetical) other language of comparable expressiveness, is not essential.

A taste of purity Before delving into the technical presentation of Pure FreshML, I encourage the reader to have a brief look at Figure 8, which shows how *normalization by evaluation* is expressed in Pure FreshML.

Normalization by evaluation is an interesting benchmark because it makes non-trivial use of names and environments. It is used by Shinwell *et al.* [23], 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 [15], and that even a manual proof of its correctness is "far from immediate" [23]. Up to a few changes and annotations, it *is* expressible in Pure FreshML.

Submitting the program in Figure 8 to the proof system results in 10 proof obligations. One such obligation arises from the use of the fresh construct (line 21). Three arise from the use of pattern matching against an abstraction pattern (lines 20, 46, and 50). Six arise from the need to establish the postcondition in the body of function *evals* (lines 41, 43, 44, 47, 51, and 53). All ten obligations are automatically discharged. This proves that the program is *pure*. That is, *normalize* denotes a (possibly

Syntactic objects

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

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

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

Figure 1: Syntax of Pure FreshML

non-terminating) pure function from λ-terms to λ-terms: the fact that it internally generates fresh atoms is not an observable side effect.

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. Cαml-style abstractions and generalized algebraic data type declarations are described only informally (§5). A couple of extended examples are presented in §6. The paper ends with discussions of related and future work (§7, §8). An early prototype implementation, together with several code samples, is available online [17].

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.* [15, 23], up to the omission of first-class functions (see §8 for a discussion). Two important features, Cαml-style abstractions and generalized algebraic data types, are omitted in this formal presentation. They are informally described in §5 and §6.

Values and patterns 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. It would be interesting to remove this restriction (§8).

Expressions 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 [5], 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 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 [15], the if construct is specialized: it compares two atoms for equality. It is possible to replace it with a general-purpose if form, while preserving the precision of the analysis, but I lack space to describe this extension.

The fresh construct generates a fresh atom. As in Pitts and Gabbay's original work [15], the atom is bound to a variable x whose scope is the expression e. In contrast, in Shinwell *et al.*'s later work [23], 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.

Function definitions 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. If desired, one function of no arguments can be distinguished as the program's entry point.

2.2 Operational semantics

Semantic values 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.

Semantic values are used in the operational semantics and in the interpretation of constraints (§3.1). They coincide with Pitts' α-terms [14, §2.4].

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

$$S/v \longrightarrow S/S(v) \tag{1}$$

$$S/\mathsf{case}\ () = v\ \mathsf{then}\ e \longrightarrow S/e \qquad \text{if } S(v) = () \tag{2}$$

$$S/\mathsf{case}\ (x_1, x_2) = v\ \mathsf{then}\ e \longrightarrow S; x_1 = w_1; x_2 = w_2/e \qquad \text{if } S(v) = (w_1, w_2) \tag{3}$$

$$S/\mathsf{case}\ K_1\, x = v\ \mathsf{then}\ e \longrightarrow S; x = w/e \qquad \text{if } S(v) = K_2\, w \text{ and } K_1 = K_2 \tag{4}$$

$$S/\mathsf{case}\ K_1\, x = v\ \mathsf{then}\ e \longrightarrow S/\mathsf{next} \qquad \text{if } S(v) = K_2\, w \text{ and } K_1 \neq K_2 \tag{5}$$

$$S/\mathsf{case}\ \langle x_1 \rangle\, x_2 = v\ \mathsf{then}\ e \longrightarrow S; a; x_1 = a; x_2 = w/e \qquad \text{if } S(v) = \langle a \rangle\, w \text{ and } a \text{ fresh for } S \tag{6}$$

$$S/\mathsf{try}\ e_1\ \mathsf{else}\ e_2 \longrightarrow S; e_2/e_1 \tag{7}$$

$$S; e/\mathsf{next} \longrightarrow S/e \tag{8}$$

$$S; F/\mathsf{next} \longrightarrow S/\mathsf{next} \qquad \text{except if the previous rule applies} \tag{9}$$

$$S; F/\mathsf{fail} \longrightarrow S/\mathsf{fail} \tag{10}$$

$$S/\mathsf{fresh}\ x\ \mathsf{in}\ e \longrightarrow S; a; x = a/e \qquad \text{if } a \text{ fresh for } S \tag{11}$$

$$S/\mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow S/e_1 \qquad \text{if } S(x_1) = a_1 \text{ and } S(x_2) = a_2 \text{ and } a_1 = a_2 \tag{12}$$

$$S/\mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow S/e_2 \qquad \text{if } S(x_1) = a_1 \text{ and } S(x_2) = a_2 \text{ and } a_1 \neq a_2 \tag{13}$$

$$S/\mathsf{let}\ x\ \mathsf{where}\ C = e_1\ \mathsf{in}\ e_2 \longrightarrow S; x.e_2/e_1 \tag{14}$$

$$S/f(v) \longrightarrow S; x_1 = S(v)/e \qquad \text{if } \mathsf{fun}\ f(x_1)\ \ldots = e \tag{15}$$

$$S; a/w \longrightarrow S/w \qquad \text{if } a \text{ fresh for } w \tag{16}$$

$$S; x.e/w \longrightarrow S; x = w/e \tag{17}$$

$$S; x = w'/w \longrightarrow S/w \tag{18}$$

$$S; e/w \longrightarrow S/w \tag{19}$$

Figure 2: Operational semantics

semantics relies on *stacks*, which, among other roles, represent a deferred substitution of semantic values for all variables in scope. This is in contrast with a more standard operational semantics based on evaluation contexts, in the style of Wright and Felleisen [27], where substitutions are not deferred.

In order to avoid deferring substitutions and to allow defining standard notions of evaluation contexts and β-reduction, an alternate approach would be to make semantic values a subset of values, and to ensure that values are stable under substitutions of a single atom for a single variable. One disadvantage of such an approach, in my opinion, would be a more complex (and perhaps confusing) treatment of values.

Stacks and configurations A *stack* S is a sequence of *frames* F (Figure 1). There are four kinds of frames, which intuitively correspond to "evaluation contexts" of depth 1, as per the following table:

frame	intuitive reading
a	fresh a in $[]$
$x.e$	let $x = []$ in e
$x = w$	let $x = w$ in $[]$
e	try $[]$ else e

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 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.

Turning values into semantic values 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 $S(v)$.

Reduction 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" [6, 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 [15, 23, 22].

T-Var
$$\Gamma \vdash x : \Gamma(x)$$

T-Unit
$$\Gamma \vdash () : \mathsf{unit}$$

T-Pair
$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2}$$

T-Sum
$$\frac{K : \tau \to \delta \qquad \Gamma \vdash v : \tau}{\Gamma \vdash K\, v : \delta}$$

T-Abs
$$\frac{\Gamma \vdash x : \mathsf{atom} \qquad \Gamma \vdash v : \tau}{\Gamma \vdash \langle x \rangle\, v : \langle \mathsf{atom} \rangle\, \tau}$$

T-Case
$$\frac{\begin{array}{c}\mathrm{dom}(p) \text{ fresh for } \Gamma \qquad \mathrm{dom}(p) = \mathrm{dom}(\Gamma') \\ \Gamma \vdash v : \tau \qquad \Gamma' \vdash p : \tau \qquad \Gamma, \Gamma' \vdash e : \tau'\end{array}}{\Gamma \vdash \mathsf{case}\ p = v\ \mathsf{then}\ e : \tau'}$$

T-Absurd
$$\Gamma \vdash \mathsf{absurd} : \tau$$

T-Next
$$\Gamma \vdash \mathsf{next} : \tau$$

T-Fail
$$\Gamma \vdash \mathsf{fail} : \tau$$

T-Try
$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{try}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

T-Fresh
$$\frac{\Gamma, x : \mathsf{atom} \vdash e : \tau}{\Gamma \vdash \mathsf{fresh}\ x\ \mathsf{in}\ e : \tau}$$

T-If
$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \mathsf{atom} \qquad \Gamma \vdash x_2 : \mathsf{atom} \\ \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash \mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

T-LetWhere
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash C \\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{where}\ C = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

T-Call
$$\frac{f : \tau_1 \to \tau_2 \qquad \Gamma \vdash v : \tau_1}{\Gamma \vdash f(v) : \tau_2}$$

T-Def
$$\frac{\begin{array}{c}f : \tau_1 \to \tau_2 \qquad x_1 : \tau_1 \vdash C_1 \\ x_1 : \tau_1, x_2 : \tau_2 \vdash C_2 \qquad x_1 : \tau_1 \vdash e : \tau_2\end{array}}{\vdash \mathsf{fun}\ f(x_1\ \mathsf{where}\ C_1) : x_2\ \mathsf{where}\ C_2 = e}$$

Figure 3: The type system (source-level objects)

Remarks The semantics is deterministic. In particular, in reduction rule 11, the choice of the fresh atom a does not matter, since its appearance in the stack frame a causes it to become bound. In reduction rule 16, the atom a, which was bound, ceases to be so, due to the destruction of the stack frame a. Fortunately, the rule is applicable only under the condition that a be fresh for w, which means that no free occurrences of a can possibly appear.

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 [23]. 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 4.7), means that valid programs are in fact pure.

When executing a valid Pure FreshML program, it is known ahead of time that nothing can go wrong, so the side condition of reduction rule 16 does not require a runtime check. As a result, all stack frames of the form a are superfluous, since their sole purpose is to enable such a runtime check. In other words, Pure FreshML *can* be efficiently implemented in terms of an uncontrolled, global fresh name generator.

Shinwell and Pitts established a "correctness of representation" result for FreshML [22, Theorem 2.3]. I believe that a Pure FreshML analogue of this result would admit a particularly direct and straightforward proof. For instance, the expressions "$\mathsf{fresh}\ x\ \mathsf{in}\ \mathsf{fresh}\ y\ \mathsf{in}\ (\langle x \rangle\, x, \langle y \rangle\, y)$" and "$\mathsf{fresh}\ x\ \mathsf{in}\ (\langle x \rangle\, x, \langle x \rangle\, x)$" both reduce, under an arbitrary stack, to the semantic value $(\langle a \rangle\, a, \langle a \rangle\, a)$, so it seems clear that these expressions are contextually equivalent.

An earlier draft of this paper presented a denotational semantics, based on nominal sets [14]. Thanks to the absence of fresh name generation as a side effect, the semantics was expressed in direct style, in contrast with Shinwell and Pitts' monadic semantics for FreshML [22]. In comparison with that earlier semantics, the operational semantics presented here is simpler, and assigns meaning to all programs, as opposed to only the valid programs. Furthermore, it seems better suited to extensions with new features such as higher-order functions and mutable state.

TS-ATOM
$$\vdash a : \mathsf{atom}$$

TS-UNIT
$$\vdash () : \mathsf{unit}$$

TS-PAIR
$$\frac{\vdash w_1 : \tau_1 \qquad \vdash w_2 : \tau_2}{\vdash (w_1, w_2) : \tau_1 \times \tau_2}$$

TS-SUM
$$\frac{K : \tau \to \delta \qquad \vdash w : \tau}{\vdash K\, w : \delta}$$

TS-ABS
$$\frac{\vdash w : \tau}{\vdash \langle a \rangle\, w : \langle \mathsf{atom} \rangle\, \tau}$$

TS-NIL
$$\epsilon \vdash \epsilon : \tau$$

TS-FRESH
$$\frac{\Gamma \vdash S : \tau}{\Gamma \vdash (S; a) : \tau}$$

TS-LET-LEFT
$$\frac{\Gamma \vdash S : \tau \qquad \Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash (S; x.e) : \tau'}$$

TS-LET-RIGHT
$$\frac{\Gamma \vdash S : \tau \qquad \vdash w : \tau'}{\Gamma, x : \tau' \vdash (S; x = w) : \tau}$$

TS-TRY
$$\frac{\Gamma \vdash S : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash (S; e) : \tau}$$

TS-CONF-EXPR
$$\frac{\Gamma \vdash S : \tau \qquad \Gamma \vdash e : \tau}{\vdash S/e\ \mathsf{ok}}$$

TS-CONF-VAL
$$\frac{\Gamma \vdash S : \tau \qquad \vdash w : \tau}{\vdash S/w\ \mathsf{ok}}$$

Figure 4: The type system (semantic objects)

2.3 Type system

I equip Pure FreshML with a conventional system of simple types [23]. 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).

Presentation The types (Figure 1) are Pitts' nominal arities [14, §2.2]. Every data constructor K carries a signature of the form $\tau \to \delta$, where δ is a data type. (The introduction of *generalized* algebraic data types is deferred to §5.) Every function f carries a signature of the form $\tau_1 \to \tau_2$.

The typing rules for values, expressions, and function definitions appear in Figure 3. Rules T-LETWHERE and T-DEF require constraints to be well-typed: this is made necessary by the fact that constraints can refer to values. The judgement $\Gamma \vdash C$ is defined in §3. It requires the values and value equations that appear within C to be well-typed under Γ.

The typing rules for semantic values, stacks, and configurations appear in Figure 4. A judgement about a stack takes the form $\Gamma \vdash S : \tau$, and states that the stack S *provides* an evaluation environment described by the type environment Γ and *expects* to receive a result of type τ. This is dual to the standard judgement $\Gamma \vdash e : \tau$, which states that expression e expects an evaluation environment described by Γ and produces a result of type τ. The two dual judgements are combined in rule TS-CONF-EXPR, whose conclusion states that S/e is a well-formed configuration.

Soundness The type system is *almost* sound: it enjoys subject reduction and *partial* progress properties. This is proven using Wright and Felleisen's standard syntactic approach [27].

This lemma states that turning a well-typed syntactic value into a semantic value always succeeds, and is a type-preserving operation.

Lemma 2.1 *Let* $\Gamma \vdash S : \tau'$ *and* $\Gamma \vdash v : \tau$. *Then,* $S(v)$ *is defined, and* $\vdash S(v) : \tau$ *holds.* ◇

Proof. The proof is by induction on the structure of v.

○ *Case* $v \equiv x$. The hypothesis $\Gamma \vdash x : \tau$ implies that x is in the domain of Γ. The hypothesis $\Gamma \vdash S : \tau'$, together with an inspection of the five typing rules for stacks, implies that every variable in the domain of Γ must be in the domain of S. Hence, S cannot be the empty stack ϵ. Two sub-cases arise.

Sub-case $S \equiv (S'; x = w)$. Then, $S(x)$ is defined as w. Since the domains of Γ and S match, and since $\Gamma \vdash x : \tau$ holds, Γ must be of the form $(\Gamma', x : \tau)$. By inverting TS-LET-RIGHT, $\Gamma \vdash S : \tau'$ gives $\vdash w : \tau$.

Sub-case $S \equiv (S'; F)$, where $F \not\equiv (x = w)$. Then, $S(x)$ is defined as $S'(x)$. Since the domains of Γ and S match, Γ must be of the form (Γ', Γ''), where Γ'' does not define x, so that $\Gamma' \vdash x : \tau$ holds. By inverting one of the four typing rules for non-empty stacks, the hypothesis $\Gamma \vdash S : \tau'$ gives $\Gamma' \vdash S : \tau''$, for some type τ''. By the induction hypothesis, we obtain that $S'(x)$ is defined and that $\vdash S'(x) : \tau$ holds.

○ *Cases* $v \equiv ()$, $v \equiv (v_1, v_2)$, and $v \equiv K\, v_1$. Immediate.

○ *Case* $v \equiv \langle x \rangle\, v_1$. By inverting T-Abs, the hypothesis $\Gamma \vdash v : \tau$ gives $\tau \equiv \langle \mathsf{atom} \rangle\, \tau_1$ and $\Gamma \vdash x : \mathsf{atom}$ and $\Gamma \vdash v_1 : \tau_1$, for some type τ_1. By the induction hypothesis, $S(x)$ is defined and $\vdash S(x) : \mathsf{atom}$ holds, which implies that $S(x)$ is an atom a. By the induction hypothesis again, $S(v_1)$ is defined and $\vdash S(v_1) : \tau_1$ holds. There follows that $S(v)$, which by definition is $\langle S(x) \rangle\, S(v_1)$, is a well-formed semantic value. Furthermore, by applying T-Abs, we find that $\vdash \langle S(x) \rangle\, S(v_1) : \tau$ holds.□

Theorem 2.2 (Subject reduction) *A well-typed configuration can reduce only to a well-typed configuration. That is,* $\vdash c_1$ *ok and* $c_1 \longrightarrow c_2$ *imply* $\vdash c_2$ *ok.* ◇

Proof. By cases over the reduction $c_1 \longrightarrow c_2$. I refer to the rule numbers in Figure 2 and use the notations in that figure.

○ *Case* (1). By inverting TS-Conf-Expr, the hypothesis $\vdash c_1$ ok yields $\Gamma \vdash S : \tau$ and $\Gamma \vdash v : \tau$. By Lemma 2.1, this implies $\vdash S(v) : \tau$, which by TS-Conf-Val implies $\vdash c_2$ ok.

○ *Cases* (2), (3), (4). Analogous to case (6).

○ *Case* (5). By inverting TS-Conf-Expr, we find $\Gamma \vdash S : \tau$. By applying T-Next, we have $\Gamma \vdash \mathsf{next} : \tau$. These imply $\vdash S/\mathsf{next}$ ok.

○ *Case* (6). By inverting T-Case and T-Abs, we find $\Gamma \vdash S : \tau$ and $\Gamma \vdash v : \langle \mathsf{atom} \rangle\, \tau_2$ and $\Gamma, x_1 : \mathsf{atom}, x_2 : \tau_2 \vdash e : \tau$. By Lemma 2.1, the first two of these imply $\vdash S(v) : \langle \mathsf{atom} \rangle\, \tau_2$. By exploiting the hypothesis $S(v) = \langle a \rangle\, w$ and by inverting TS-Abs, we find $\vdash w : \tau_2$. By applying TS-Fresh once and TS-Let-Right twice, we successively derive:

$$\Gamma \vdash (S; a) : \tau$$
$$\Gamma, x_1 : \mathsf{atom} \vdash (S; a; x_1 = a) : \tau$$
$$\Gamma, x_1 : \mathsf{atom}, x_2 : \tau_2 \vdash (S; a; x_1 = a; x_2 = w) : \tau$$

This implies $\vdash S; a; x_1 = a; x_2 = w/e$ ok.

○ *Case* (7). By inverting TS-Conf-Expr and T-Try, we find $\Gamma \vdash S : \tau$ and $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. By applying TS-Try, the first and last of these imply $\Gamma \vdash (S; e_2) : \tau$. This implies $\vdash (S; e_2)/e_1$ ok.

○ *Case* (8). By inverting TS-Conf-Expr and TS-Try, we find $\Gamma \vdash S : \tau$ and $\Gamma \vdash e : \tau$, which imply $\vdash S/e$ ok.

○ *Case* (9). By inverting TS-Conf-Expr and one of the four typing rules for non-empty stacks, we get $\Gamma \vdash S : \tau$, for some arbitrary Γ and τ. By applying T-Next, we have $\Gamma \vdash \mathsf{next} : \tau$. These imply $\vdash S/\mathsf{next}$ ok.

○ *Case* (10). Analogous to case (9).

○ *Case* (11). Analogous to case (6).

○ *Case* (12). By inverting TS-Conf-Expr and T-If, we get, among other hypotheses, $\Gamma \vdash S : \tau$ and $\Gamma \vdash e_1 : \tau$. These imply $\vdash S/e_1$ ok.

○ *Case* (13). Analogous to case (12).

○ *Case* (14). By inverting TS-Conf-Expr and T-LetWhere, we get $\Gamma \vdash S : \tau_2$ and $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. By applying TS-Let-Left, the first and last of these imply $\Gamma \vdash (S; x.e_2) : \tau_1$. There follows $\vdash (S; x.e_2)/e_1$ ok.

○ *Case* (15). By inverting TS-Conf-Expr and T-Call, we get $\Gamma \vdash S : \tau_2$ and $f : \tau_1 \to \tau_2$ and $\Gamma \vdash v : \tau_1$. By Lemma 2.1, these imply $\vdash S(v) : \tau_1$. By applying TS-Let-Right, we derive $\Gamma, x : \tau_1 \vdash (S; x = S(v)) : \tau_2$. By recalling the hypothesis $\mathsf{fun}\ f(x_1)\ \ldots = e$ and inverting T-Def, we get $x_1 : \tau_1 \vdash e : \tau_2$, which *a fortiori* implies $\Gamma, x_1 : \tau_1 \vdash e : \tau_2$. There follows $\vdash (S; x_1 = S(v))/e$ ok.

○ *Case* (16). By inverting TS-Conf-Val and TS-Fresh, we get $\Gamma \vdash S : \tau$ and $\vdash w : \tau$. There follows $\vdash S/w$ ok.

○ *Case* (17). By inverting TS-Conf-Val and TS-Let-Left, we get $\Gamma \vdash S : \tau$ and $\Gamma, x : \tau' \vdash e : \tau$ and $\vdash w : \tau'$. By applying TS-Let-Right, the first and last of these imply $\Gamma, x : \tau' \vdash (S; x = w) : \tau$. There follows $\vdash (S; x = w)/e$ ok.

○ *Case* (18). By inverting TS-Conf-Val and TS-Let-Right, we get $\Gamma \vdash S : \tau$ and $\vdash w : \tau$. There follows $\vdash S/w$ ok.

○ *Case* (19). By inverting TS-Conf-Val and TS-Try, we get $\Gamma \vdash S : \tau$ and $\vdash w : \tau$. There follows $\vdash S/w$ ok. □

Theorem 2.3 (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.* ◇

Proof. For configurations of the form S/e, by cases over e.

○ *Case* $e \equiv v$. By inverting TS-CONF-EXPR, we get $\Gamma \vdash S : \tau$ and $\Gamma \vdash v : \tau$. By Lemma 2.1, this implies that $S(v)$ is defined. So, reduction rule 1 is applicable.

○ *Case* $e \equiv \mathsf{case}\ p = v\ \mathsf{then}\ e_1$. Four sub-cases arise, depending on the structure of p. We deal with the sub-case where $p \equiv \langle x_1 \rangle\, x_2$; the other sub-cases are analogous. By inverting T-CASE and T-ABS, we find $\Gamma \vdash S : \tau$ and $\Gamma \vdash v : \langle \mathsf{atom} \rangle\, \tau_2$. By Lemma 2.1, these imply that $S(v)$ is defined and that $\vdash S(v) : \langle \mathsf{atom} \rangle\, \tau_2$ holds. This implies that $S(v)$ is of the form $\langle a \rangle\, w$, for some atom a, which, without loss of generality, we can take to be fresh for S. So, reduction rule 6 is applicable.

○ *Case* $e \equiv \mathsf{absurd}$. This is one of the two kinds of blocked configurations allowed by the theorem's statement.

○ *Case* $e \equiv \mathsf{next}$. If S is the empty stack ϵ, then S/e is a result. Otherwise, one of reduction rule 8 and reduction rule 9 is applicable.

○ *Case* $e \equiv \mathsf{fail}$. If S is the empty stack ϵ, then S/e is a result. Otherwise, reduction rule 10 is applicable.

○ *Case* $e \equiv \mathsf{try}\ e_1\ \mathsf{else}\ e_2$. Reduction rule 7 is applicable.

○ *Case* $e \equiv \mathsf{fresh}\ x\ \mathsf{in}\ e_1$. Reduction rule 11 is applicable.

○ *Case* $e \equiv \mathsf{if}\ x_1 = x_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. By inverting TS-CONF-EXPR and T-IF, we get $\Gamma \vdash S : \tau$ and $\Gamma \vdash x_1 : \mathsf{atom}$ and $\Gamma \vdash x_2 : \mathsf{atom}$. By Lemma 2.1, these imply that $S(x_1)$ and $S(x_2)$ are defined and have type atom, which means that they are atoms a_1 and a_2. So, one of reduction rule 12 and reduction rule 13 is applicable.

○ *Case* $e \equiv \mathsf{let}\ x\ \mathsf{where}\ C = e_1\ \mathsf{in}\ e_2$. Reduction rule 14 is applicable.

○ *Case* $e \equiv f(v)$. By inverting TS-CONF-EXPR and T-CALL, we get $\Gamma \vdash S : \tau$ and $\Gamma \vdash v : \tau_1$. By Lemma 2.1, these imply that $S(v)$ is defined. So, reduction rule 15 is applicable.

For configurations of the form S/w, by cases over S.

○ *Case* $S \equiv \epsilon$. Then, S/w is a result.

○ *Case* $S \equiv (S; a)$. If a is fresh for w, then reduction rule 16 is applicable. Otherwise, this is one of the two kinds of blocked configurations allowed by the theorem's statement.

○ *Case* $S \equiv (S; x.e_1)$. Reduction rule 17 is applicable.

○ *Case* $S \equiv (S; x = w_1)$. Reduction rule 18 is applicable.

○ *Case* $S \equiv (S; e_1)$. Reduction rule 19 is applicable. □

The statement of Theorem 2.3 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

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

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

I use the following sugar:

$$\begin{array}{r@{\ }l} s_1 \setminus s_2 & \text{stands for } s_1 \cap \neg s_2 \\ \textit{false} & \text{stands for } \mathbb{A} = \emptyset \\ \textit{true} & \text{stands for } \emptyset = \emptyset \\ s_1 \subseteq s_2 & \text{stands for } (s_1 \cap \neg s_2) = \emptyset \\ s_1 = s_2 & \text{stands for } (s_1 \subseteq s_2) \wedge (s_2 \subseteq s_1) \\ s_1 \mathrel{\#} s_2 & \text{stands for } (s_1 \cap s_2) = \emptyset \end{array}$$

Set expressions denote sets of atoms, that is, elements of the Boolean algebra $\mathcal{P}(\mathbb{A})$, the powerset of the set of atoms $\mathbb{A}$. The most interesting form of set expression is the application of the mathematical function *fa* to a value v. (For now, only the *fa* function is made available, but other functions are introduced in §5.) Set expressions can then be built up using the standard set-theoretic connectives for the empty set of atoms, the full set of atoms, intersection, union, and negation. Constraints are conjunctions of *atomic constraints*: set emptiness (or non-emptiness) assertions and value equations.

Despite its name, Pure FreshML is an *impure* language in the sense that there *is* one side effect: non-termination. For this reason, I follow standard practice and allow constraints to depend on variables, and, more generally, on values, but never on arbitrary expressions.

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.

Interpretation 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 $\mathcal{P}(\mathbb{A})$. The set-theoretic connectives and the set-theoretic atomic constraints are interpreted in the algebra $\mathcal{P}(\mathbb{A})$. 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 (that is, when some valuation satisfies C) and $C_1 \Vdash C_2$ when C_1 entails C_2 (that is, when every valuation that satisfies C_1 also satisfies C_2).

3.2 Eliminating value equations

I first eliminate value equations, that is, I reduce general entailment and satisfiability problems down to problems that involve no value equations. The reduction is sound, and incomplete.

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 reduced to satisfiability problems by exploiting the following facts:

$$\begin{array}{c} C \Vdash C_1 \wedge C_2 \text{ if and only if } C \Vdash C_1 \text{ and } C \Vdash C_2 \\ C \Vdash s = \emptyset \text{ if and only if not } \vdash C \wedge s \neq \emptyset \\ C \Vdash s \neq \emptyset \text{ if and only if not } \vdash C \wedge s = \emptyset \end{array}$$

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.

Step 1 The first step can be viewed as a closure computation, defined by the following rules:

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

A rule $C \rightarrow C'$ means: if the conjunct C exists, add the conjunct C'. The process is iterated until a fixed point is reached. In practice, it can be implemented efficiently in terms of first-order unification of values. 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 equation $\langle x_1 \rangle\, v_1 = \langle x_2 \rangle\, v_2$ does have consequences, which can be stated as follows [25]: first, the atom x_1 is not in the support of the value $\langle x_2 \rangle\, v_2$; second, the values v_1 and v_2 are images of one another modulo swapping of the atoms x_1 and x_2. Because the second statement cannot be expressed as a constraint, I pretend that the equation $\langle x_1 \rangle\, v_1 = \langle x_2 \rangle\, v_2$ has no consequences. Thus, information is lost in step 2 when this equation is dropped.

Step 2 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. In practice, I expect equations of the form $v_1 = v_2$, where neither v_1 nor v_2 is a variable, to rarely arise. Indeed, they appear only when a value is constructed and immediately deconstructed, a pattern that seems unlikely to occur, at least in programs written by humans.

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.

Step 1 First, applications of *fa* are reduced:

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

As a result, only applications of the form $\textit{fa}(x)$ remain. It is clear that this simplification process preserves the interpretation of constraints.

Step 2 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.

On "purity" 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 [15] 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.

This rule is generalized when functions other than *fa* are introduced (§5). These new functions also take the value $\emptyset$ at certain types, and it is important for the system to know about it.

On "definite impurity" and absurdity 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 5). In short, negative hypotheses of the form $X \neq \emptyset$ are required in order to establish absurdity.

Conversely, by the independence property of negative constraints, a negative hypothesis cannot help establish a positive goal (unless it in fact establishes absurdity). So, if one is willing to accept the postulate that the user should never write non-trivial code in a context where absurd is permitted, then nothing is lost by *not* exploiting the negative hypotheses when trying to establish a goal *other* than absurdity. This remark is practically important, as it greatly reduces the number of problems that must be presented to the SAT solver.

On incompleteness 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

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

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 [10] 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 $\mathcal{P}(\mathbb{A})$.* $\diamond$

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 $\mathbb{B}$.* $\diamond$

Proof. This result is known in the case of positive constraints [10]. So, let us consider a simple constraint of the form $C \wedge s \neq 0$, where C is a conjunction of positive atomic constraints.

Assume that $C \wedge s \neq 0$ is satisfiable over $\mathcal{P}(\mathbb{A})$. Then, there exists a valuation ϕ (a mapping of the variables to subsets of $\mathbb{A}$) that satisfies it. We have, in particular, $\phi(s) \neq \emptyset$, so there exists an atom $a \in \phi(s)$. Let us now define a mapping f of $\mathcal{P}(\mathbb{A})$ onto $\mathbb{B}$ as follows: for every $A \in \mathcal{P}(\mathbb{A})$, $f(A)$ is 1 if $a \in A$ and 0 otherwise. It is clear that f is a homomorphism, that is, f preserves all of the Boolean connectives. As a result, if a positive (atomic or non-atomic) constraint is satisfied over $\mathcal{P}(\mathbb{A})$ by ϕ, then it is also satisfied over $\mathbb{B}$ by $f \circ \phi$. That is, $f \circ \phi$ satisfies C. Furthermore, by construction, $f \circ \phi$ satisfies $s \neq 0$: indeed, $a \in \phi(s)$ implies $f(\phi(s)) = 1$. We have proved that $C \wedge s \neq 0$ is satisfiable over $\mathbb{B}$.

Conversely, pick an arbitrary $a \in \mathbb{A}$. Then, $\mathbb{B}$ is isomorphic to the subalgebra $\{\emptyset, \{a\}\}$ of $\mathcal{P}(\mathbb{A})$. Thus, any constraint (simple or not simple) that is satisfiable over $\mathbb{B}$ is also satisfiable over $\mathcal{P}(\mathbb{A})$. $\square$

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. Today, moderate-size instances of this problem are easily solved using off-the-shelf tools such as Chaff and its variants [11]. I should point out that the problems that I generate are small: for instance, the sample programs in §6 give rise to problems whose conjunctive normal forms exhibit at most 20 variables and 80 clauses.

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 5). In order to establish the soundness of the proof system, judgements on stacks and configurations are also required (Figure 6).

Before explaining the judgements, it is worth stressing the distinction between two distinct kinds of freshness requirements. In Figures 5 and 6, a premise of the form "x fresh for $\ldots$" is a standard *meta-theoretic* freshness requirement, bearing on the *meta-variable* x. Such a requirement, found in all type systems or proof systems, is satisfied by ensuring that "all names are distinct" in the program, and can safely be ignored by the casual reader. On the other hand, formulas of the form "*fa*(x) # $\ldots$" are *constraints* specific to this paper, bearing on the *value denoted by* x. They are explicit hypotheses or goals and are eventually transmitted to the decision procedure for entailment problems.

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

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

$$\textsc{Case}\qquad \frac{\begin{array}{c}\mathrm{dom}(p) \text{ fresh for } \Delta, H, v, P \qquad \Delta \vdash \{H'\}\, p\, \{P'\} \\ \Delta, \mathrm{dom}(p) \vdash \{H \wedge H' \wedge p = v\}\, e\, \{P \wedge P'\}\end{array}}{\Delta \vdash \{H\}\ \mathsf{case}\ p = v\ \mathsf{then}\ e\, \{P\}} \qquad\qquad \textsc{Absurd}\qquad \frac{H \Vdash \mathit{false}}{\Delta \vdash \{H\}\ \mathsf{absurd}\, \{P\}}$$

$$\textsc{Next}\qquad \Delta \vdash \{H\}\ \mathsf{next}\, \{P\} \qquad\qquad \textsc{Fail}\qquad \Delta \vdash \{H\}\ \mathsf{fail}\, \{P\} \qquad\qquad \textsc{Try}\qquad \frac{\begin{array}{c}\Delta \vdash \{H\}\, e_1\, \{P\} \\ \Delta \vdash \{H\}\, e_2\, \{P\}\end{array}}{\Delta \vdash \{H\}\ \mathsf{try}\ e_1\ \mathsf{else}\ e_2\, \{P\}}$$

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

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

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

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

Figure 5: The proof system (source-level objects)

$$\textsc{Stk-Nil}\qquad \vdash \epsilon\, \{\mathit{true}\} \qquad\qquad \textsc{Stk-Fresh}\qquad \frac{\vdash S\, \{P\}}{\vdash (S; a)\, \{P \wedge a \# \mathit{fa}(\cdot)\}}$$

$$\textsc{Stk-Let-Left}\qquad \frac{\begin{array}{c}\Delta = \mathrm{dom}(S) \qquad x \text{ fresh for } \Delta, P \\ \vdash S\, \{P\} \qquad \Delta, x \vdash \{S \wedge C\}\, e\, \{P\}\end{array}}{\vdash (S; x.e)\, \{\lambda x.C\}}$$

$$\textsc{Stk-Let-Right}\qquad \frac{x \text{ fresh for } P \qquad \vdash S\, \{P\}}{\vdash (S; x = w)\, \{P\}} \qquad\qquad \textsc{Stk-Try}\qquad \frac{\begin{array}{c}\Delta = \mathrm{dom}(S) \qquad \vdash S\, \{P\} \\ \Delta \vdash \{S\}\, e\, \{P\}\end{array}}{\vdash (S; e)\, \{P\}}$$

$$\textsc{Conf-Expr}\qquad \frac{\begin{array}{c}\Delta = \mathrm{dom}(S) \qquad \vdash S\, \{P\} \\ \Delta \vdash \{S\}\, e\, \{P\}\end{array}}{\vdash S/e\ \mathsf{ok}} \qquad\qquad \textsc{Conf-Val}\qquad \frac{\vdash S\, \{P\} \qquad S \Vdash P(w)}{\vdash S/w\ \mathsf{ok}}$$

Figure 6: The proof system (semantic objects)

Expressions 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. The use of triples of a precondition, a program fragment, and a postcondition dates back to Hoare. More recently, Honda and Yoshida [7] have developed a proof system whose triples take the same form as mine.

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.

In an algorithmic reading of the definition, all four components (Δ, H, e, and P) should be considered inputs. The output of the algorithm consists in the proof obligations carried by the leaves of the derivation (VALUE, ABSURD, CALL).

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. (The symbol $\equiv$ means "is of the form".)

Rule ABSURD emits a proof obligation that requires the current hypothesis to be inconsistent. This ensures that the absurd statement is unreachable.

Rules NEXT and FAIL state that the exceptions next and fail can be used in an arbitrary context, with no proof obligation whatsoever. Rule TRY requires both branches to satisfy the Hoare triple. The fact that the second branch is executed only if the first branch raises next is not reflected in the current proof system. When the try construct is used to encode surface-level case constructs, this means that each branch of a case construct is analyzed in isolation, without regard for the patterns that guard previous branches.

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. Within e_2, nothing else is known about x. Thus, an appropriate choice of C is important. As noted earlier, I do not attempt to *infer* a strongest postcondition for e_1.

Rule CALL emits two proof obligations. One checks that f's precondition is satisfied by the actual argument v. The other checks that the postcondition P of the call statement is implied by f's postcondition. In the second premise, I use the notation $H \Vdash P_1 \Rightarrow P_2$, where H is a constraint and P_1, P_2 are predicates, to denote the fact that, under hypothesis H, predicate P_1 is stronger than P_2. This can also be written $H \Vdash \forall x.P_1(x) \Rightarrow P_2(x)$, or, equivalently, $H \wedge P_1(x) \Vdash P_2(x)$, for a fresh x.

Function definitions Rule DEF states that the body of a function must be checked under the precondition C_1 and postcondition C_2 that were provided by the user. It also defines the notations "$\mathrm{pre}(f)$" and "$\mathrm{post}(f)$" used in rule CALL. One interesting point is that $\mathrm{post}(f)$ contains not only C_2, but also $\mathit{fa}(x_2) \subseteq \mathit{fa}(x_1)$. This means that, at every call site, the support of the result can be assumed to be a subset of the support of the argument. This assumption comes "for free". It is, in fact, a consequence of the fact that toplevel functions must have empty support. It is justified by Lemma 4.4.

Judgements about function definitions take the form $\vdash fd$. The entire program is accepted by the proof system if and only if every single function definition is.

Stacks and configurations In order to establish the soundness of the proof system, the system must be extended to stacks and configurations. This is done in Figure 6. The judgement $\vdash S\,\{P\}$ states that the stack S expects a result that satisfies the predicate P.

In these rules and in the soundness proof, I use constraints that mix syntactic elements (values) and semantic elements (semantic values). For instance, rule STK-FRESH states that the stack $(S; a)$ expects a result that is fresh for the atom a. I also view a stack S as a constraint, obtained as the conjunction of all equations of the form $x = w$ found within S. For instance, in the last premise of rule STK-LET-LEFT, the precondition under which e is checked is $S \wedge C$.

4.2 Soundness

Lemma 4.1 (Precondition Strengthening) *$H' \Vdash H$ and $\Delta \vdash \{H\}\,e\,\{P\}$ imply $\Delta \vdash \{H'\}\,e\,\{P\}$.* ◇

Proof. By a straightforward induction over the derivation of $\Delta \vdash \{H\}\,e\,\{P\}$. □

Lemma 4.2 (Postcondition Weakening) *$H \Vdash P \Rightarrow P'$ and $\Delta \vdash \{H\}\,e\,\{P\}$ imply $\Delta \vdash \{H\}\,e\,\{P'\}$.* ◇

Proof. By a straightforward induction over the derivation of $\Delta \vdash \{H\}\,e\,\{P\}$. □

Lemma 4.3 (Environment Widening) *$\Delta' \supseteq \Delta$ and $\Delta \vdash \{H\}\,e\,\{P\}$ imply $\Delta' \vdash \{H\}\,e\,\{P\}$.* ◇

Proof. By a straightforward induction over the derivation of $\Delta \vdash \{H\}\,e\,\{P\}$. In case FRESH, note that widening Δ leads to strengthening the constraint $H \wedge \mathit{fa}(x) \# \mathit{fa}(\Delta)$, so that applying Lemma 4.1 to the second premise, in conjunction with the induction hypothesis, yields the result. □

The following lemma states that, if an expression e is accepted at all by the proof system, then the system is in fact sufficiently strong to prove that the atoms that appear free in the result of evaluating e must also appear free in the initial evaluation environment. In other words, evaluating a provably correct expression e does not cause any new atoms to appear.

Lemma 4.4 (No Atoms Made Up) *$\Delta \vdash \{H\}\,e\,\{P\}$ implies $\Delta \vdash \{H\}\,e\,\{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$.* ◇

Proof. By induction over the derivation of $\Delta \vdash \{H\}\,e\,\{P\}$. I adopt the notations of Figure 5. I exploit the implicit hypothesis that the free variables of e form a subset of Δ.

○ *Case* VALUE. One first shows that, when the free variables of v form a subset of Δ, the constraint $\mathit{fa}(v) \subseteq \mathit{fa}(\Delta)$ is valid, that is, satisfied by every valuation. The proof of this fact is by structural induction over v, and is immediate. As a result of this fact, VALUE's premise $H \Vdash P(v)$ implies $H \Vdash P(v) \wedge \mathit{fa}(v) \subseteq \mathit{fa}(\Delta)$. By applying VALUE, we get $\Delta \vdash \{H\}\,v\,\{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$.

○ *Case* CASE. As above, the constraint $\mathit{fa}(v) \subseteq \mathit{fa}(\Delta)$ is valid. Let Δ' stand for $(\Delta, \mathrm{dom}(p))$. Then, applying the induction hypothesis to CASE's third premise yields

$$\Delta' \vdash \{H \wedge H' \wedge p = v\}\,e\,\{P \wedge P' \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta')\}$$

We will now show that, under $p = v$, the predicate $P' \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta')$ entails $\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)$. To do so, we consider two sub-cases.

Sub-case $p \equiv \langle x_1 \rangle\, x_2$. By inverting ABSTRACTION-PATTERN, we find that P' is $\mathit{fa}(x_1) \# \mathit{fa}(\cdot)$. As a result, the predicate

$$P' \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta')$$

can be written

$$\mathit{fa}(x_1) \# \mathit{fa}(\cdot) \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta, x_1, x_2)$$

which is equivalent to

$$\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta, x_2) \setminus \mathit{fa}(x_1)$$

Furthermore, the constraint $p = v$ entails $\mathit{fa}(x_2) \setminus \mathit{fa}(x_1) = \mathit{fa}(v)$, so that, under $p = v$, the above predicate entails

$$\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta) \cup \mathit{fa}(v)$$

which is equivalent to

$$\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)$$

This ends the first sub-case.

Sub-case $p \not\equiv \langle x_1 \rangle\, x_2$. A case analysis over p shows that $p = v$ entails $\mathit{fa}(\mathrm{dom}(p)) = \mathit{fa}(v)$, whence $\mathit{fa}(\Delta') = \mathit{fa}(\Delta)$. As a result, under $p = v$, the predicate $\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta')$ entails $\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)$. This ends the second sub-case.

We now apply Lemma 4.2, which yields

$$\Delta' \vdash \{H \wedge H' \wedge p = v\}\, e\, \{P \wedge P' \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$$

There only remains to apply CASE to conclude

$$\Delta \vdash \{H\}\, \mathsf{case}\ p = v\ \mathsf{then}\ e\, \{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$$

○ *Cases* ABSURD, NEXT, FAIL. Immediate.

○ *Cases* TRY, IF. Apply the induction hypothesis and conclude.

○ *Case* FRESH. Analogous to the first sub-case of case CASE.

○ *Case* LETWHERE. Applying the induction hypothesis to the second premise yields

$$\Delta, x \vdash \{H \wedge C \wedge \mathit{fa}(x) \subseteq \mathit{fa}(\Delta)\}\, e_2\, \{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta, x)\}$$

It is clear that $\mathit{fa}(x) \subseteq \mathit{fa}(\Delta)$ entails $\mathit{fa}(\Delta, x) = \mathit{fa}(\Delta)$, so, by applying Lemma 4.2 to the above, we get:

$$\Delta, x \vdash \{H \wedge C \wedge \mathit{fa}(x) \subseteq \mathit{fa}(\Delta)\}\, e_2\, \{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$$

By applying LETWHERE, we conclude:

$$\Delta \vdash \{H\}\, \mathsf{let}\ x\ \mathsf{where}\ C = e_1\ \mathsf{in}\ e_2\, \{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$$

○ *Case* CALL. Let the definition of the function f be

$$\mathsf{fun}\ f(x_1\ \mathsf{where}\ C_1) : x_2\ \mathsf{where}\ C_2 = e$$

Then, by inverting DEF, we find

$$\mathrm{post}(f) = \lambda(x_1, x_2).(C_2 \wedge \mathit{fa}(x_2) \subseteq \mathit{fa}(x_1))$$

which means that $\mathrm{post}(f)(v, \cdot)$ entails $\mathit{fa}(\cdot) \subseteq \mathit{fa}(v)$. As in previous cases, the constraint $\mathit{fa}(v) \subseteq \mathit{fa}(\Delta)$ is valid, so $\mathrm{post}(f)(v, \cdot)$ entails $\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)$. By applying CALL, we conclude:

$$\Delta \vdash \{H\}\, f(v)\, \{P \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\} \qquad \square$$

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 4.5 (Subject Reduction) *A valid configuration can reduce only to a valid configuration. That is, $\vdash c_1$ ok and $c_1 \longrightarrow c_2$ imply $\vdash c_2$ ok.* ◇

Proof. By cases over the reduction $c_1 \longrightarrow c_2$. I refer to the rule numbers in Figure 2 and use the notations in that figure. In every case, Δ stands for $\mathrm{dom}(S)$.

○ *Case* (1). By inverting CONF-EXPR and VALUE, we get $\vdash S\,\{P\}$ and $S \Vdash P(v)$. The latter implies $S \Vdash P(w)$, where $w = S(v)$. By applying VALUE and CONF-VAL, we find $\vdash S/w$ ok.

○ *Case* (3). By inverting CONF-EXPR, CASE, and OTHER-PATTERN, we get

$$\vdash S\,\{P\}$$
$$x_1, x_2 \text{ fresh for } \Delta, v, P$$
$$\Delta, x_1, x_2 \vdash \{S \wedge (x_1, x_2) = v\}\, e\, \{P\}$$

Due to the hypothesis $S(v) = (w_1, w_2)$, and to the fact that x_1 and x_2 are fresh for Δ and v, the constraint $S \wedge (x_1, x_2) = v$ is equivalent to $S; x_1 = w_1; x_2 = w_2$. Thus, by Lemma 4.1, we have

$$\Delta' \vdash \{S'\}\, e\, \{P\}$$

where Δ' stands for Δ, x_1, x_2 and S' stands for $S; x_1 = w_1; x_2 = w_2$. Furthermore, thanks to the hypothesis $\vdash S\,\{P\}$ and to the fact that x_1 and x_2 are fresh for P, we have

$$\vdash S'\,\{P\}$$

By applying CONF-EXPR, we finally obtain $\vdash S'/e$ ok.

○ *Cases* (2), (4), (5). Analogous to case (3).

○ *Case* (6). Analogous to a combination of cases (3) and (11).

○ *Case* (7). By inverting CONF-EXPR and TRY, we get

$$\vdash S\,\{P\}$$
$$\Delta \vdash \{H\}\,e_1\,\{P\}$$
$$\Delta \vdash \{H\}\,e_2\,\{P\}$$

By applying STK-TRY and CONF-EXPR, we obtain $\vdash (S; e_2)/e_1$ ok.

○ *Case* (8). By inverting CONF-EXPR and STK-TRY, we get

$$\vdash S\,\{P\}$$
$$\Delta \vdash \{H\}\,e\,\{P\}$$

By applying CONF-EXPR, we obtain $\vdash S/e$ ok.

○ *Case* (9). By inverting CONF-EXPR, we get

$$\vdash (S; F)\,\{P\}$$

By inverting one of STK-FRESH, STK-LET-LEFT, or STK-LET-RIGHT, we get

$$\vdash S\,\{P'\}$$

for some predicate P'. Now, by applying NEXT, we have

$$\Delta \vdash \{S\}\,\mathsf{next}\,\{P'\}$$

Finally, by applying CONF-EXPR, we obtain $\vdash S/\mathsf{next}$ ok.

○ *Case* (10). Analogous to case (9).

○ *Case* (11). By inverting CONF-EXPR and FRESH, we get

$$\vdash S\,\{P\}$$
$$x \text{ fresh for } \Delta, P$$
$$\Delta, x \vdash \{S \wedge \mathit{fa}(x) \# \mathit{fa}(\Delta)\}\,e\,\{P \wedge \mathit{fa}(x) \# \mathit{fa}(\cdot)\}$$

Let S' stand for $S; a; x = a$. Because x is fresh for Δ, we have

$$S' \Vdash S$$

Because a is fresh for S, we have

$$x = a \Vdash \mathit{fa}(x) \# \mathit{fa}(\Delta)$$

Furthermore, we have

$$x = a \Vdash \mathit{fa}(x) \# \mathit{fa}(\cdot) \Rightarrow a \# \mathit{fa}(\cdot)$$

Thus, by applying Lemma 4.1 and Lemma 4.2, we obtain

$$\Delta, x \vdash \{S'\}\,e\,\{P \wedge a \# \mathit{fa}(\cdot)\}$$

Besides, by applying STK-FRESH and STK-LET-RIGHT, we find

$$\vdash S'\,\{P \wedge a \# \mathit{fa}(\cdot)\}$$

Finally, by applying CONF-EXPR, we obtain $\vdash S'/e$ ok.

○ *Case* (12). By inverting Conf-Expr and If, we get

$$\vdash S\,\{P\}$$
$$\Delta \vdash \{S \wedge \mathit{fa}(x_1) = \mathit{fa}(x_2)\}\, e_1\, \{P\}$$

Thanks to the hypotheses $S(x_1) = a_1$, $S(x_2) = a_2$, and $a_1 = a_2$, we have

$$S \Vdash \mathit{fa}(x_1) = \mathit{fa}(x_2)$$

Thus, by Lemma 4.1, we have

$$\Delta \vdash \{S\}\, e_1\, \{P\}$$

By applying Conf-Expr, we finally obtain $\vdash S/e_1$ ok.

○ *Case* (13). Analogous to case (12).

○ *Case* (14). By inverting Conf-Expr and LetWhere, we get

$$\vdash S\,\{P\}$$
$$x \text{ fresh for } \Delta, P$$
$$\Delta \vdash \{S\}\, e_1\, \{\lambda x.C\}$$
$$\Delta, x \vdash \{S \wedge C \wedge \mathit{fa}(x) \subseteq \mathit{fa}(\Delta)\}\, e_2\, \{P\}$$

By applying Stk-Let-Left, we obtain

$$\vdash S'\,\{(\lambda x.C) \wedge \mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta)\}$$

where S' stands for $S; x.e_2$. Note that $\Delta = \mathrm{dom}(S) = \mathrm{dom}(S')$. Furthermore, because the stacks S and S' give rise to the same constraint, we have

$$\Delta \vdash \{S'\}\, e_1\, \{\lambda x.C\}$$

By Lemma 4.4, this implies

$$\Delta \vdash \{S'\}\, e_1\, \{(\lambda x.C) \wedge (\mathit{fa}(\cdot) \subseteq \mathit{fa}(\Delta))\}$$

By applying Conf-Expr, we finally obtain $\vdash S'/e_1$ ok.

○ *Case* (15). By inverting Conf-Expr and Call, we get

$$\vdash S\,\{P\}$$
$$S \Vdash \mathrm{pre}(f)(v)$$
$$S \Vdash \mathrm{post}(f)(v, \cdot) \Rightarrow P$$

By inverting Def, we get

$$x_1 \vdash \{C_1\}\, e\, \{\lambda x_2.C_2\}$$
$$\mathrm{pre}(f) = \lambda x_1.C_1$$
$$\mathrm{post}(f) = \lambda(x_1, x_2).(C_2 \wedge \mathit{fa}(x_2) \subseteq \mathit{fa}(x_1))$$

Without loss of generality, we pick x_1 fresh for Δ, v, P. Let S' stand for $S; x_1 = S(v)$. By applying Stk-Let-Right, we obtain

$$\vdash S'\,\{P\}$$

Because x_1 is fresh for v, we have $S'(v) = S(v)$, which implies $S' \Vdash x_1 = v$. Furthermore, because x_1 is fresh for Δ, we have $S' \Vdash S$. As a result, we have

$$\begin{aligned} S' &\Vdash S \wedge x_1 = v \\ &\Vdash \mathrm{pre}(f)(v) \wedge x_1 = v \\ &\Vdash \mathrm{pre}(f)(x_1) \\ &= C_1 \end{aligned}$$

Hence, by Lemma 4.1, we have

$$x_1 \vdash \{S'\}\, e\, \{\lambda x_2.C_2\}$$

By Lemma 4.4, this implies

$$x_1 \vdash \{S'\}\, e\, \{\lambda x_2.(C_2 \wedge \mathit{fa}(x_2) \subseteq \mathit{fa}(x_1))\}$$

that is,

$$x_1 \vdash \{S'\}\, e\, \{\mathrm{post}(f)(x_1, \cdot)\}$$

Again, under S', x_1 and v coincide, so this can be written:

$$x_1 \vdash \{S'\}\, e\, \{\mathrm{post}(f)(v, \cdot)\}$$

By Lemma 4.2 and Lemma 4.3, this implies

$$\Delta, x_1 \vdash \{S'\}\, e\, \{P\}$$

By applying CONF-EXPR, we finally obtain $\vdash S'/e$ ok.

○ *Case* (16). By inverting CONF-VAL and STK-FRESH, we get

$$\vdash S\, \{P\}$$
$$S \Vdash (P \wedge a \# \mathit{fa}(\cdot))(w)$$

In particular, we have

$$S \Vdash P(w)$$

By applying CONF-VAL, we obtain $\vdash S/w$ ok.

○ *Case* (17). By inverting CONF-VAL and STK-LET-LEFT, we get

$$x \text{ fresh for } \Delta, P$$
$$\vdash S\, \{P\}$$
$$\Delta, x \vdash \{S \wedge C\}\, e\, \{P\}$$
$$S \Vdash (\lambda x.C)(w)$$

The last fact above implies

$$S' \Vdash C$$

where S' stands for $S; x = w$. Furthermore, because x is fresh for Δ, we have

$$S' \Vdash S$$

By Lemma 4.1, these imply

$$\Delta, x \vdash \{S'\}\, e\, \{P\}$$

By applying STK-LET-RIGHT, we obtain

$$\vdash S'\, \{P\}$$

By applying CONF-EXPR, we finally get $\vdash S'/e$ ok.

○ *Case* (18). By inverting CONF-VAL and STK-LET-RIGHT, we get

$$x \text{ fresh for } P$$
$$\vdash S\, \{P\}$$
$$S; x = w' \Vdash P(w)$$

The first and last of these imply

$$S \Vdash P(w)$$

By applying CONF-VAL, we get $\vdash S/w$ ok.

○ *Case* (19). By inverting CONF-VAL and STK-TRY, we get

$$\vdash S\, \{P\}$$
$$S; e \Vdash P(w)$$

The last of these implies

$$S \Vdash P(w)$$

By applying CONF-VAL, we get $\vdash S/w$ ok. □

Theorem 4.6 (Progress) *A valid, irreducible configuration is a result.* ◇

Proof. By Theorem 2.3, only two cases need be examined and ruled out.

○ *Case* S/absurd. By inverting CONF-EXPR and ABSURD, we get $S \Vdash \mathit{false}$. Yet, a stack S, when viewed as a constraint, is a satisfiable constraint, since S is its own satisfying valuation. A contradiction follows.

○ *Case* $S; a/w$, where a occurs free in w. By inverting CONF-VAL and STK-FRESH, we get, in particular:

$$S \Vdash (P \wedge a \mathrel{\#} \mathit{fa}(\cdot))(w)$$

This implies that the ground constraint $a \mathrel{\#} \mathit{fa}(w)$ is valid, that is, a is fresh for w. This contradicts our hypothesis that a occurs free in w. □

Corollary 4.7 (Soundness) *A valid configuration cannot go wrong.* ◇

Proof. By Theorem 2.2 and Theorem 4.5, a valid configuration can reduce only to a valid configuration. By Theorem 4.6, a valid, irreducible configuration is a result. Together, these facts imply that a reduction sequence out of a valid configuration must either diverge or converge to a result. In other words, such a sequence cannot go wrong, that is, lead to an irreducible configuration that is not a result. □

5 Extensions

I now informally present Cαml-style abstractions and generalized algebraic data types, which, for the sake of simplicity, I omitted in the formal presentation of the proof system.

5.1 Cαml-style abstractions

Presentation I have advocated elsewhere [16] that FreshML's binary abstraction construct $\langle x\rangle\, e$ is too limited for many practical uses. Fresh Objective Caml's more general construct $\langle e_1\rangle\, e_2$ is also inconvenient, because it requires structuring every abstraction as a *pair* whose left-hand component e_1 holds *all* binding occurrences of atoms (and *only* them) and whose right-hand component e_2 corresponds to the scope of the abstraction. This construct cannot express such typical idioms as lists of bindings, environments, etc. For this reason, Cαml [16] offers a richer binding specification language, which Pure FreshML can adopt. In principle, the syntax of values becomes:

$$v ::= x \mid () \mid (v, v) \mid K\, v \mid \langle v\rangle \mid \mathsf{inner}\, v \mid \mathsf{outer}\, v$$

In my prototype implementation, the angle brackets $\langle\cdot\rangle$ as well as the inner and outer keywords appear only in algebraic data type declarations, and become implicitly attached to data constructors. This allows omitting them in the actual syntax of values.

Two kinds of values are distinguished, called "expressions" and "patterns" in [16]. (Better terminology would be needed.) Inside the former, an occurrence of an atom is regarded as a *free* occurrence. Inside the latter, an occurrence of an atom is interpreted as a *binding* occurrence. The abstraction former $\langle\cdot\rangle$, which is now unary, expects a "pattern" and constructs an "expression". Conversely, the keywords inner and outer expect an "expression" and construct a "pattern". They serve to end an enclosing abstraction, that is, they mean: "as far as the current abstraction is concerned, there are no binding occurrences of atoms below this point". In addition, inner (resp. outer) indicates that the value that follows lies *within* (resp. *outside*) the scope of the abstraction.

Due to space constraints, it is impossible to repeat here a full explanation of Cαml's binding specification language. The reader is referred to the existing paper [16] as well as to the examples contained in the present paper (§6). For instance, the definition of the data constructor *Lam* on line 3 of Figure 8 illustrates the Cαml idiom for an abstraction that would be expressible in FreshML. The definition of the data constructor *L* (line 7), together with the definition of the "pattern" type *env* (line 14), shows how to define an abstraction that binds a statically unknown number of names. More explanations are provided in §6.

$$\mathit{fa}(\langle v\rangle) \to \mathit{outer}(v) \cup (\mathit{inner}(v) \setminus \mathit{bound}(v))$$

$$\begin{array}{l}
\text{for } f \in \{\mathit{fa}, \mathit{inner}, \mathit{outer}, \mathit{bound}\},\\
\qquad\qquad\qquad\qquad\qquad f(()) \to \emptyset\\
\qquad\qquad\qquad\qquad f((v_1, v_2)) \to f(v_1) \cup f(v_2)\\
\qquad\qquad\qquad\qquad\qquad f(K\,v) \to f(v)
\end{array}$$

$$\begin{array}{l}
\text{for } f \in \{\mathit{inner}, \mathit{outer}, \mathit{bound}\},\\
\qquad\qquad\qquad\qquad f(\langle v\rangle) \text{ is undefined}
\end{array}$$

$$\begin{aligned}
\mathit{fa}(\mathsf{inner}\,v) &\to \mathit{fa}(v)\\
\mathit{fa}(\mathsf{outer}\,v) &\to \mathit{fa}(v)\\
\mathit{inner}(\mathsf{inner}\,v) &\to \mathit{fa}(v)\\
\mathit{inner}(\mathsf{outer}\,v) &\to \emptyset\\
\mathit{outer}(\mathsf{inner}\,v) &\to \emptyset\\
\mathit{outer}(\mathsf{outer}\,v) &\to \mathit{fa}(v)\\
\mathit{bound}(\mathsf{inner}\,v) &\to \emptyset\\
\mathit{bound}(\mathsf{outer}\,v) &\to \emptyset
\end{aligned}$$

Figure 7: Reducing applications of set functions to non-variables

Changes to Pure FreshML As far as Pure FreshML is concerned, the introduction of Cαml's binding specification language has relatively little impact. I focus, in the following, on the changes made to the decision procedure and proof system.

The introduction of a richer binding specification language affects the way in which the support of a value is computed. In particular, the support of an abstraction $\langle v\rangle$ satisfies the equation:

$$\mathit{fa}(\langle v\rangle) = \mathit{outer}(v) \cup (\mathit{inner}(v) \setminus \mathit{bound}(v))$$

Here, v is a value of "pattern" type. Its support is computed in terms of three auxiliary functions, whose informal meaning is as follows: *inner*(v) (resp. *outer*(v)) is the combined support of the sub-values of v that appear below an inner (resp. outer) keyword, while *bound*(v) is the set of atoms that have a binding occurrence in v (*not* below an inner or outer keyword).

The syntax of set expressions (§3.1) is extended to include *inner*(v), *outer*(v), and *bound*(v) for all values v of "pattern" type, while *fa* remains available at all types.

The simplification rules that define the meaning of *fa* are modified so as to reflect the above equation, and supplemented with new simplification rules for *inner*, *outer*, and *bound*. All updated rules appear in Figure 7.

The technique that was used to eliminate applications of *fa* to variables is extended to also deal with applications of *outer*, *inner*, and *bound*. Again, the type of the variable influences this step. In particular, if x has type atom, then both *inner*(x) and *outer*(x) are empty, and *bound*(x) equals *fa*(x). As a more subtle example, if x has type *env* as defined on line 14 of Figure 8, then *inner*(x) must be empty.

The universal law $f(x) \subseteq \mathit{fa}(x)$, where f is any set function (here, one of *inner*, *outer*, and *bound*) is reflected by introducing an explicit constraint on the set variables that stand for $f(x)$ and *fa*(x).

The proof rules are almost unchanged. The only notable change is in rule ABSTRACTION-PATTERN (Figure 5), which becomes:

ABSTRACTION-PATTERN'
$\Delta \vdash \{\mathit{bound}(x) \mathrel{\#} \mathit{fa}(\Delta)\}\ \langle x\rangle\ \{\mathit{bound}(x) \mathrel{\#} \mathit{fa}(\cdot)\}$

In short, the set *fa*(x_1) of ABSTRACTION-PATTERN, which denotes a single atom, is replaced with the set *bound*(p), which possibly denotes zero, one, or more atoms. This reflects the fact that an abstraction now binds a set of atoms at once.

5.2 Generalized algebraic data types

Presentation So far, I have used traditional algebraic data types, where every data constructor carries a signature of the form $\tau \to \delta$ (§2.3). In practice, it is sometimes useful to attach *assertions*, also known as *guards*, with data constructors. This is done by letting every data constructor K carry a signature of the more general form:

$$(x : \tau \,\mathsf{where}\, C) \to \delta$$

Here, the identifier x is bound in the constraint C. In type-theoretic terms, the data constructor K now carries a dependent pair of a value of type τ, referred to as x, and a proof (not represented at runtime, of course) that x satisfies the constraint C. For example, the type *context* (line 8 of Figure 9) is a generalized algebraic data type. Its definition is explained further on (§6.2).

Changes to Pure FreshML The changes to the proof system are simple. An application of K to a value v now gives rise to the proof obligation $H \Vdash [x \mapsto v]C$, where H is the current hypothesis. Conversely, matching against the pattern $K\,p$ augments H with the new conjunct $[x \mapsto p]C$.

There is a slight catch, though. The semantics of FreshML dictates that, when an abstraction is deconstructed via pattern matching, its bound atoms are replaced with fresh atoms (see reduction rule 6). If, because of this renaming, the property C was broken, then the proof system would be unsound.

Could such a thing happen? In fact, not when C has a single free variable x, as above. It is not difficult to prove that, if a predicate $\lambda x.C$ is true of a value v, then it is also true of any renaming of v. One *can* get into trouble, however, when one allows guards to refer to *multiple* variables, to which *distinct* renamings could be applied. For instance, my prototype implementation allows data constructors to carry multiple arguments, some of which can be nested inside abstractions, so one might attempt to write:

```
type dangerous =
| Danger of x: atom × ⟨ y: atom ⟩ where free(x) = free(y)
```

This is meaningless. Matching against the pattern *Danger*(x, y) leaves x unmodified but renames y to a fresh atom, so the property that x equals y cannot possibly be preserved.

For this reason, such a declaration must be ruled out. I have not formalized how this is done, but the informal rule is simple: a guard must not mix variables that originate in distinct abstractions.

6 Illustration

I now discuss two small but non-trivial example programs that are accepted by the proof system: normalization by evaluation (Figure 8) and conversion to A-normal form (Figure 9).

The concrete syntax is that of my prototype implementation. As explained earlier (§2.1), what is shown here is *surface* syntax. It is translated down to Pure FreshML before being passed on to the proof system. The details of the translation are omitted.

The concrete syntax uses the keyword *free* for *fa*.

6.1 Normalization by evaluation

I first explain how Shinwell *et al.*'s benchmark [23] is adapted to Pure FreshML. There are three main changes, which I now review.

Simulating first-class functions First, because my proof system does not yet have first-class functions, I have defunctionalized [18] the code. That is, I have replaced every first-class function with a *closure*—a data structure formed of a tag and a tuple of the function's free variables.

Here, the effect of defunctionalization is that the data constructor *L* carries data—a triple of an environment, an atom, and a term—instead of a first-class function. These three components correspond to the three free variables of the function that was carried by *L* in the original code [23, figure 7, line 27]. Where a first-class function was applied in the original code [23, figure 7, lines 16 and 30], explicit use is made of the closure—which boils down to a recursive call to *evals* (lines 22 and 51).

In order to simplify things a little, I have replaced the type unit → *sem* [23, figure 7, line 8] with just *sem*. This affects the termination of the algorithm, but makes essentially no difference as far as proof obligations are concerned. Being faithful to the original code would have required introducing one more data type and one more auxiliary function, with no pedagogic gain.

```
type lam =
  | Var of atom
  | Lam of ⟨ atom × inner lam ⟩
  | App of lam × lam

type sem =
  | L of ⟨ env × atom × inner lam ⟩
  | N of neu

type neu =
  | V of atom
  | A of neu × sem

type env binds =
  | ENil
  | ECons of env × atom × outer sem

fun reify accepts s produces t =
  case s of
  | L (env, y, t) →
      fresh x in
      Lam (x, reify (evals (ECons (env, y, N (V (x))), t)))
  | N (n) →
      reifyn (n)
  end

fun reifyn accepts n produces t =
  case n of
  | V (x) →
      Var (x)
  | A (n, d) →
      App (reifyn (n), reify (d))
  end

fun evals accepts env, t produces v
where free(v) ⊆ outer(env) ∪ (free(t) \ bound(env)) =
  case t of
  | Var (x) →
      case env of
      | ENil →
          N (V (x))
      | ECons (tail, y, v) →
          if x = y then v
          else evals (tail, t) end
      end
  | Lam (x, t) →
      L (env, x, t)
  | App (t1, t2) →
      case evals (env, t1) of
      | L (cenv, x, t) →
          evals (ECons (cenv, x, evals (env, t2)), t)
      | N (n) →
          N (A (n, evals (env, t2)))
      end
  end

fun eval accepts t produces s =
  evals (ENil, t)

fun normalize accepts t produces u =
  reify (eval (t))
```

Figure 8: A sample program: normalization by evaluation

Specifying the binding structure of environments and closures Second, I have made essential use of Cαml's "binding specification" language. The definition of the type *lam* is identical to Shinwell *et al.*'s, modulo differences in notation. The key novelty is in the definition of the type *env*, which I have made a "pattern type", in Cαml parlance [16]—this is indicated by the *binds* keyword on line 14 of Figure 8. This means that, in an environment of the form

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

the atoms $x_1, \ldots, x_n$ are considered as *binding occurrences*. Thus, by definition, $\mathit{bound}(\mathit{env})$ denotes the set of atoms $\{x_1, \ldots, x_n\}$, which one would usually refer to as the *domain* of the environment.

According to the *outer* keyword in the definition of *ECons* (line 16), the semantic values $v_1, \ldots, v_n$ are considered to lie *outside* the scope of these atoms. Thus, $\mathit{outer}(\mathit{env})$ denotes the set of atoms $\mathit{fa}(v_1) \cup \ldots \cup \mathit{fa}(v_n)$, which one might refer to as the *image* of the environment.

What is the scope of the atoms $x_1, \ldots, x_n$? According to the *inner* keyword in the definition of L (line 7), within a closure of the form

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

the atoms in $\mathit{bound}(\mathit{env})$, as well as the atom x, are considered bound within the λ-term t. In particular, this implies:

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

That is, the support of the closure v includes the *image* of its environment *env*, as well as the atoms that appear free in its body t and are neither in the *domain* of *env* nor the formal argument x.

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 $L(\mathit{env}, x, t)$. The fact that x is "fresh" means, in particular, that x is not in the support of *env*, which by definition includes $\mathit{outer}(\mathit{env})$. By exploiting the above displayed equation, one finds that x is not in the support of $L(\mathit{env}, x, t)$, as desired. This fact is proved automatically by the conservative decision procedure of §3.

The proof obligation on line 46 is interesting because it corresponds, in part, to the obligation that FreshML 2000 was not able to automatically discharge [23, 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 of these require exploiting a non-trivial property of *evals*, which I now discuss.

Specifying *evals*' behavior with respect to support 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):

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

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.

An alternative way of providing this information to the proof system would be to have *evals* accept an *abstraction* of type $\langle \mathit{env} \times \mathsf{inner}\ \mathit{lam} \rangle$, instead of a pair of type $\mathit{env} \times \mathit{lam}$. Then, no explicit postcondition would be required: the support of *evals*' result would be simply the support of its argument. This alternative style can seem attractive, but is less efficient if abstractions are blindly "freshened" when deconstructed. I come back to this issue in §8.

This property is easily proved correct. Because *evals* is recursive, the proof is "by induction"—that is, the property is exploited in its own proof. This might seem surprising, because there is no guarantee that *evals* terminates. This approach is sound, because the property is a *partial* correctness assertion: it is a statement about the result of *evals*, should it terminate.

6.2 Conversion to A-normal form

Figure 9 defines the abstract syntax of a λ-calculus equipped with let and if constructs and gives an algorithm that converts arbitrary terms to A-normal form, as defined by Flanagan *et al.* [5]. A-normal

```
type term =
  | Var of atom
  | Lambda of ⟨ atom × inner term ⟩
  | App of term × term
  | Let of ⟨ atom × outer term × inner term ⟩
  | If of term × term × term

type context binds =
  | CEmpty
  | CLet of x: atom × inner t: term × c: context
          where free(t) # free(x) ∪ bound(c)
  | CCompose of c1: context × c2: context
              where inner(c1) # bound(c2)

type closure =
  | Clo of ⟨ context × inner term ⟩

fun fill accepts clo produces u =
  let Clo (c, t) = clo in
  case c of
  | CEmpty →
      t
  | CLet (x, t1, c2) →
      Let (x, t1, fill (Clo (c2, t)))
  | CCompose (c1, c2) →
      fill (Clo (c1, fill (Clo (c2, t))))
  end

fun norm accepts t produces u =
  fill (split (t, false))

fun split accepts t, mode produces clo =
  case t of
  | Var (_) →
      Clo (CEmpty, t)
  | Lambda (x, t) →
      Clo (CEmpty, Lambda (x, norm (t)))
  | App (t1, t2) →
      let Clo (c1, u1) = split (t1, true) in
      let Clo (c2, u2) = split (t2, true) in
      let clo = Clo (CCompose (c1, c2), App (u1, u2)) in
      valueify (clo, mode)
  | Let (x, t1, t2) →
      let Clo (c1, u1) = split (t1, false) in
      let Clo (c2, u2) = split (t2, mode) in
      Clo (CCompose (c1, CLet (x, u1, c2)), u2)
  | If (t1, t2, t3) →
      let Clo (c1, u1) = split (t1, true) in
      let clo = Clo (c1, If (u1, norm (t2), norm (t3))) in
      valueify (clo, mode)
  end

fun valueify accepts clo, mode produces clo =
  if mode then
    let Clo (c, t) = clo in
    fresh x in
    Clo (CCompose (c, CLet (x, t, CEmpty)), Var (x))
  else
    clo
  end
```

Figure 9: A sample program: conversion to A-normal form

form requires operators of applications, operands of applications, and conditions of if constructs to be values, and forbids nesting of let constructs towards the left.

Flanagan *et al.* provide a rather subtle conversion algorithm, expressed in continuation-passing style. I was surprised to find that this algorithm, once defunctionalized and translated to the input language of my prototype implementation, requires only single-atom abstractions, as opposed to Cαml-style abstractions, and gives rise to only a handful of proof obligations, all of which are trivial. This algorithm is probably expressible in FreshML 2000 [15].

In order to make things more interesting, I present a different algorithm, expressed in direct style (Figure 9). I would say that this algorithm is conceptually more straightforward than Flanagan *et al.*'s—there are no continuations, no first-class functions, or defunctionalized versions thereof. Yet, it requires advanced use of Cαml-style abstractions and of generalized algebraic data types, and gives rise to 17 proof obligations, many of which are non-trivial. This is because the algorithm makes explicit use of *contexts*: the central function, *split*, produces a pair of a context—a sequence of let definitions that are being floated out—and a residual term. It is interesting that an arguably more natural algorithm should require a significantly more powerful proof system!

Contexts The abstract syntax of contexts is simple. It would be written, on paper, as follows:

$$c ::= [] \mid \mathsf{let}\ x = t\ \mathsf{in}\ c \mid c_1[c_2]$$

Intuitively, a context is just an ordered list of bindings of the form

$$\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = t_n\ \mathsf{in}\ []$$

In the following, I refer to the set $\{x_1, \ldots, x_n\}$ as the *domain* of such a context. In the code, things will be set up so that the domain of *c* is referred to as *bound*(c).

The first two productions in the above grammar would be sufficient to generate all lists of bindings. The third production, which denotes list concatenation, is conceptually redundant, but allows constant time composition of contexts.

The term $c[t]$ obtained by filling context *c* with term *t* would be defined, on paper, as follows:

$$\begin{aligned} [][t] &= t \\ (\mathsf{let}\ x = t_1\ \mathsf{in}\ c)[t_2] &= \mathsf{let}\ x = t_1\ \mathsf{in}\ c[t_2] \\ (c_1[c_2])[t] &= c_1[c_2[t]] \end{aligned}$$

This corresponds to function *fill* (line 18). Context filling is, by design, a *capturing* operation: any atom that occurs free in *t* and is in the domain of *c* becomes bound in the term $c[t]$. It is important to note that the atoms that form the domain of *c* occur *free* in *c*, that is, they are members of *fa*(c). They become bound *only* when *c* is filled with a term.

Representing closures Several functions (*fill*, *split*, and *valueify*) accept or return pairs of a context *c* and a term *t*, where *t* is to be viewed as "within the context *c*". One cannot fuse the two by forming the term $c[t]$ right away because the inductive definition of *split* requires individual access to *c* and *t*. They are eventually fused when *norm*, the algorithm's main entry point, invokes *fill* (line 30).

What does it mean for *t* to be viewed as "within the context *c*"? The answer is, even though one has not yet filled the hole and formed $c[t]$, one *promises* to do so in the future, so that the atoms in the domain of *c can* be considered bound in the pair (c, t).

I formalize this intuition by wrapping *c* and *t* together in a *closure* (line 15), that is, a Cαml-style abstraction, where the atoms in the domain of *c* are declared to be bound within *t*.

Representing contexts Because a context defines a set of atoms that are bound by the *closure* abstraction, the type *context* must be a "pattern" type, in Cαml parlance [16]. This is indicated by the *binds* keyword on line 8. The three data constructor declarations for *CEmpty*, *CLet*, and *CCompose* reflect the abstract syntax of contexts that was given earlier. Two non-trivial aspects, which I now explain, are the use of the *inner* keyword (line 10) and of a guard (lines 11 and 13).

Since *term* is a Cαml "expression" type, while *context* is a "pattern" type, the *term* component in the declaration of *CLet* must be preceded with one of the *inner* or *outer* keywords [16], so as to indicate whether this term lies *inside* or *outside* the scope of the abstractions in which contexts participate (here, closures).

Which of the two keywords is appropriate here? Suppose I construct the closure

$$clo = Clo(c, t)$$

where c is a context of the form

$$\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = t_n\ \mathsf{in}\ []$$

Within this closure, should the terms $t_1, \ldots, t_n$ lie *inside* or *outside* of the scope of the atoms $x_1, \ldots, x_n$? Aha, that's a trick question. One answer is, *neither*. Considering how the let forms are nested, each t_i should lie within the scope of $\{x_1, \ldots, x_{i-1}\}$.

If neither keyword is appropriate, are we out of luck? Is Cαml's binding specification language too crude for this application? No—there is a way out. I use the *inner* keyword, thus *pretending* that each t_i lies within the scope of $\{x_1, \ldots, x_n\}$. Then, I add a side condition (line 11) stating that t_i contains no occurrence of the atoms $\{x_i, \ldots, x_n\}$. The end effect is exactly what was needed! The side condition carried by *CCompose* (line 13) serves the same purpose.

I don't know how general this trick really is. I believe it is quite interesting, and could also be useful in the setting of a proof assistant, should one attempt to mechanize, in a nominal style, proofs that involve nested contexts.

The algorithm Once appropriate definitions of the types *context* and *closure* are made, the code is straightforward. In short, *fill* fills a context *c* with a term *t*, producing a term. *norm* accepts a term and produces its A-normal form. *split* accepts a pair of a term t_1 and a Boolean flag *mode* and produces a closure $Clo(c_2, t_2)$ such that t_1 is semantically equivalent to $c_2[t_2]$ and, if *mode* is *true*, then t_2 is a value. *valueify* accepts a pair of a closure $Clo(c_1, t_1)$ and a Boolean flag *mode* and produces a closure $Clo(c_2, t_2)$ such that $c_1[t_1]$ is semantically equivalent to $c_2[t_2]$ and, if *mode* is *true*, then t_2 is a value. If *mode* is *true*, *valueify* defines t_2 to be a fresh variable *x* and floats the binding "let $x = t_1$ in []" up into the context (lines 56 and 57). It is essential to have *valueify* return a closure, as opposed to a pair of a context and a term. Otherwise, the proof system would think that *x* escapes the scope of the fresh construct that created it.

The code gives rise to 17 proof obligations, all of which are successfully and automatically discharged.

It is remarkable that there are no visible assertions in the code. Of course, this is an illusion, since the numerous explicit uses of *Clo* are really annotations.

Introducing an error Imagine that, on line 46, the programmer is confused and writes

```
Clo (CCompose (c1, CCompose (c2, CLet (x, u1, CEmpty))), u2)
```

That is, she constructs the context $c1[c2[\mathsf{let}\ x = u1\ \mathsf{in}\ []]]$ instead of $c1[\mathsf{let}\ x = u1\ \mathsf{in}\ c2]$.

This incorrect program is rejected. The current prototype implementation produces the following error message:

```
File "anf-direct.fml", line 46, characters 25-60:
I am unable to prove that the following hypotheses:
inner(c1) ⊆ free(c1)
bound(c1) ⊆ free(c1)
bound(c2) ⊆ free(c2)
inner(c2) ⊆ free(c2)
free(?closure__1) = (inner(c2) ∪ free(u2)) \ bound(c2)
free(?closure) = (free(u1) ∪ inner(c1)) \ bound(c1)
free(t) = free(t1) ∪ free(t2) \ free(x)
bound(c2) # free(x) ∪ free(t) ∪ free(t1) ∪ free(t2)
          ∪ free(c1) ∪ free(u1) ∪ free(?closure)
          ∪ free(?closure__1)
free(?closure__1) ⊆ free(t2)
bound(c1) # free(x) ∪ free(t) ∪ free(t1) ∪ free(t2)
          ∪ free(?closure)
```

```
free(?closure) ⊆ free(t1)
free(x) # free(t)
entail the goal:
inner(c2) # free(x)
The reason why I am attempting to prove this assertion is...
File "anf-direct.fml", line 46, characters 25-60:
It is part of the invariant for data constructor CCompose.
```

The list of hypotheses is rather difficult to decipher. (The names *?closure* and *?closure__1* stand for the results of the two recursive calls to *split*. They are generated during the translation of the surface language down to the kernel language described in this paper.) The proof system complains that, under a certain set of hypotheses, it cannot prove ***inner***$(c2)$ # ***free***(x). (This proof obligation corresponds to the guard of the right-hand *CCompose*.) This means that the atom *x* could appear free in the context *c2*. This is true: *c2* was constructed out of *t2*, which can contain free occurrences of *x*. For this reason, $c2[\mathsf{let}\ x = u1\ \mathsf{in}\ []]$ is not a well-formed context.

The quality of this error message could hopefully be improved. The point, for now, is that this subtle programming error, which a standard type system would not have caught, is detected by the proof system.

Note that the sets ***free***$(mode)$, ***outer***$(c1)$, ***outer***$(c2)$, etc. are not mentioned in any of the hypotheses. By examining the types of *mode*, *c1*, and *c2*, the system can tell that these sets are empty. (This was discussed in §3.3.) This knowledge can be necessary for the proof obligations to go through, and helps reduce visual clutter.

7 Related work

This paper was inspired by Pitts and Gabbay's work on static "freshness inference" for FreshML [15]. 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*. It does not attempt to do any kind of inference besides the simple type inference performed by the underlying type system. For this reason, explicit assertions must sometimes be provided at let constructs. I did initially attempt to infer an approximation of the support of values and expressions, but I found that this approach was much more complex and not worth the trouble.

The design of a dependent type system for an impure programming language was pioneered by Xi [28]. The key insight that constraints can be dependent only on values, as opposed to arbitrary computations, is exploited here.

Pašalić and Linger [13] 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 [4] explore a similar approach in the programming language ATS. These are interesting ideas, but I believe that the nominal programming style supported by FreshML is more natural and appealing than a de Bruijn-based approach.

Schürmann *et al.*'s ∇-calculus [19] 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—which, in this case, also means that object-level terms are lexically well-formed. 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. This is quite impressive, but, again, I find nominal encodings much more direct than higher-order abstract syntax encodings. The price to pay for the simpler, nominal approach is the need to hand-code substitution functions, or to carry explicit environments around.

MetaOCaml relies on environment classifiers [24] 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. This approach seems coarser than that followed in the present paper, but lends itself better to type inference techniques [2].

Nanevski's calculus $\nu^{\Box}$ [12] 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.

Kim, Yi, and Calcagno [9] present a meta-programming language equipped with a type system that uses rows to keep track of the free names of each code fragment. The language is not hygienic, however—a code fragment can refer to the name "x" in a context where no such name was ever introduced.

A line of works by Jackson *et al.* [8, 26, 3] rely on a SAT solver to detect bugs in software. A finite approximation of the procedure's behavior is encoded as a formula in first-order relational logic. It is then conjoined with the procedure's precondition and with the negation of the procedure's postcondition, so as to look for executions that violate the procedure's specification. The resulting formula in first-order relational logic is translated down, under a finite bound on the size of its models, to propositional logic, and handed to a SAT solver. This approach appears effective at finding bugs, but cannot prove their absence.

8 Future work

Many features must be added in order to turn Pure FreshML into a realistic meta-programming language. Here are a few:

- *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 [15] 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.

- *Exceptions*. Their addition should be unproblematic, provided that every function declares which exceptions it can raise and (if necessary) provides postconditions for exceptional exits.

- *Primitive operations*. The language should provide sets of atoms, maps over atoms, etc. The proof system should keep precise track of all operations over these data structures.

- *Multiple sorts of atoms*. Distinguishing multiple sorts of atoms is easy [16], useful, and, in the setting of Pure FreshML, provides extra freshness assumptions for free: two atoms of distinct sorts are automatically known to be fresh for one another.

- *Polymorphism*. Type polymorphism, sort polymorphism, and parameterized algebraic data types are important features that I have left aside until now. Their combination with Cαml-style algebraic data types could raise non-trivial issues.

- *Non-linear patterns*. As noted by Pitts and Gabbay [15, §5.2], non-linear patterns sometimes offer an elegant way of avoiding an explicit renaming. It would be interesting to extend the dynamic semantics and the proof system with direct support for them.

- *Safe non-freshening*. The nominal approach to abstract syntax has been criticized for its runtime cost. *Freshening*, that is, automatically replacing an abstraction's bound atoms with fresh atoms when that abstraction is inspected, is expensive. Furthermore, it can be unnecessary: sometimes, there simply is no risk of inadvertent capture. I believe that a Pure FreshML compiler could detect many such situations and produce efficient code (by performing no freshening) without sacrificing safety.

- *Typed abstract syntax*. It is well-known that generalized algebraic data types [29] allow reflecting the typing rules of a simply-typed object language into the meta-language. Combining this technique with Pure FreshML would lead to a meta-programming language that can only construct *lexically well-formed* and *well-typed* object program fragments.

A mid-term goal is to design a realistic meta-programming language on top of Pure FreshML. In order to ensure interoperability with existing libraries, it would be compiled down to Objective Caml, using some of the techniques developed for Cαml [16].

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] C. Calcagno, E. Moggi, and W. Taha. ML-like inference for classifiers. In *European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 79–93. Springer Verlag, 2004.

[3] G. Dennis, F. Change, and D. Jackson. Modular verification of code with SAT. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2006.

[4] 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.

[5] 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.

[6] 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.

[7] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *International ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 191–202, Aug. 2004.

[8] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis (ISSTA)*, Aug. 2000.

[9] I.-S. Kim, K. Yi, and C. Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 257–268, 2006.

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

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, July 2001.

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, Dec. 1998.

[19] 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.

[20] 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.

[21] M. R. Shinwell. Fresh O'Caml: nominal abstract syntax for the masses. In *ACM Workshop on ML*, Sept. 2005.

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

[23] 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.

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

[25] C. Urban, A. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

[26] M. Vaziri and D. Jackson. Checking heap-manipulating procedures with a constraint solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*. Springer Verlag, Apr. 2003.

[27] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.

[28] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Dec. 1998.

[29] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 224–235, Jan. 2003.