Noname manuscript No.
(will be inserted by the editor)

The Locally Nameless Representation

Arthur Charguéraud

Received: date / Accepted: date

Abstract This paper provides an introduction to the locally nameless approach to the representation of syntax with variable binding, focusing in particular on the use of this technique in formal proofs. First, we explain the benefits of representing bound variables with de Bruijn indices while retaining names for free variables. Then, we explain how to describe and manipulate syntax in that form, and show how to define and reason about judgments on locally nameless terms.

1 Introduction

Most programming languages, type systems and logical systems make use of variables. Many different techniques are available to represent syntax with variable bindings in a given programming language or in a given formal theory. This paper focuses on one particular representation of bindings, called the *locally nameless* representation. It has been successfully used to mechanize soundness proofs of type systems, properties of the semantics of λ-calculi, and correctness proofs of program transformations [Leroy, 2007, Aydemir et al., 2008, Charguéraud, 2009]. This representation has also been shown useful in the implementation of type checkers and proof checkers, among which Coq [Coq Development Team, 2009], Lego [Luo and Pollack, 1992], Isabelle [Nipkow et al., 2002], HOL 4 [Norrish and Slind, 2007] and Epigram [Altenkirch et al., 2005].

The locally nameless representation relies on de Bruijn indices to represent bound variables but uses names to represent free variables. Such a mixed syntax allows for a very simple implementation of substitution and β-reduction. By featuring a unique representation of terms, it avoids traditional issues related to α-conversion. In the same time, it allows for a reasoning style fairly close to the style in which pencil-and-paper proofs are conventionally carried out. The purpose of this paper is to provide a thorough

Arthur Charguéraud
INRIA Rocquencourt
Domaine de Voluceau
Rocquencourt - B.P. 105
78153 Le Chesnay, France
E-mail: arthur.chargueraud@inria.fr

introduction to the locally nameless representation, explaining how it works, why it works, and how to use it in formal reasoning.

The introduction of the locally nameless representation is *not* a contribution of this paper. The possibility for combining de Bruijn indices with names was in fact mentioned by de Bruijn [1972] in his founding paper. It has been used in early implementations of proof assistants, starting with Huet's *Constructive Engine* [1989] and Paulson's Isabelle proof system [1986]. In the context of formal proofs, Gordon [1993] appears to be the first to have used the locally nameless representation, although he used it only as a basis for building an interface with named λ-terms rather than directly reasoning on locally nameless syntax. Later work by Gordon and Melham [1996] also uses locally nameless terms as a model for an abstract axiomatic representation of named terms.

Pollack [2006] has more recently emphasized the benefits of the locally nameless representation in the context of the POPLMark challenge [Aydemir et al., 2005], building on his experience of formalizing Pure Type Systems with a representation featuring distinguished bound named variables and free named variables [McKinna and Pollack, 1993]. The locally nameless representation was first experimented by Leroy [2007] on the POPLMark Challenge. Further investigations and larger-scale case studies using this representation were then conducted by Aydemir et al. [2008].

The first contribution of this paper is a thorough and complete description of the locally nameless representation. We start by motivating the introduction of this technique through an analysis of the strengths and drawbacks of related approaches to representing bindings. We then present the operations involved for manipulating locally nameless syntax, and discuss their implementation. We also give formal statements of the key properties verified by these operations and explain when such properties need to be exploited in formal reasoning.

The second contribution of this paper is a detailed introduction to carrying out formal reasoning on programming languages and type systems described in locally nameless style. We first recall how to define judgments on λ-terms using a cofinite quantification technique introduced by the author and his co-authors [Aydemir et al., 2008]. We then show how to formally prove standard properties about these judgments such as type soundness, proof of confluence and semantic preservation.

The third contribution of this paper consists in the generalization of the locally nameless representation to advanced forms of binding structures. We explain how to support multiple binders, recursive binders, mutually-recursive binders and pattern matching structures, both for linear and non-linear patterns. Supporting these ingredients is essential to the formalization of the syntax and semantics of realistic programming languages.

2 The locally nameless representation

There exist many possibilities for representing variable bindings. Our goal is not to cover all of them, but only to discuss representations that are closely-related to the locally nameless representation. (The paper by Aydemir et al. [2008] contains a survey of binding techniques.) Most issues related to variable bindings can be studied on a language as simple as the pure λ-calculus. Thus, only the syntax of λ-terms is considered throughout the core of the paper. Support for more advanced binding structures is investigated afterwards (§7).

2.1 Named representations: raw terms and quotiented terms

The most common representation of λ-terms relies on the use of names: each abstraction and each variable bear a *name*. The syntax of *raw named terms* is described by the following grammar.

$$t \quad := \quad \mathsf{var}\, x \quad | \quad \mathsf{abs}\, x\, t \quad | \quad \mathsf{app}\, t\, t$$

The objects from this grammar are called raw terms because they are not isomorphic to λ-terms. For example, the two raw terms "$\mathsf{abs}\, x\, (\mathsf{var}\, x)$" and "$\mathsf{abs}\, y\, (\mathsf{var}\, y)$" are two different objects, although the two λ-terms "$\lambda x.\, x$" and "$\lambda y.\, y$" should be considered equal because the theory of λ-calculus identifies terms that are α-equivalent. Due to the mismatch between raw terms and λ-terms, there are pieces of reasoning from λ-calculus textbooks that cannot be formalized using raw terms.

In order to obtain a representation of terms truly isomorphic to λ-terms, we need to build a quotient structure, quotienting the set of raw terms with respect to alpha-equivalence. This construction based on a quotient corresponds very closely to the of presentation from standard textbooks on λ-calculus.

In practice, though, working formally with a quotient structure is not that straightforward. In order to define a function or a relation on λ-terms, we need to first define it on raw terms, then show it compatible with α-equivalence, and finally lift it to the quotient structure. For instance, if f is a unary function on terms in the named representation, then, for f to be accepted as a definition on λ-terms, we must prove that, for any two alpha-equivalent terms t_1 and t_2, the two applications $f(t_1)$ and $f(t_2)$ yield α-equivalent results. Lifting definitions to the quotient structure is typically long and tedious. Fortunately, a lot of this work can be automated. For example, Urban's *nominal package* [2008] aims at factorizing and automating definitions and proofs about data types involving binders. Yet, at this time, there are still a number of advanced binding structures that are not supported by the nominal package.

2.2 The locally named representation

The locally nameless representation is closely related to the *locally named* representation, which has been extensively developed by McKinna and Pollack [1993]. This representation syntactically distinguishes between bound variables and free variables. Bound variables are represented using a name, written x. Free variables, also called *parameters*, are represented using another kind of names, written p. Abstractions, which always bind "bound variables", carry a bound variable name. The grammar of locally named terms can thus be described as follows.

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs}\, x\, t \quad | \quad \mathsf{app}\, t\, t$$

The main interest of the locally named representation is that a bound name and a free name can never be confused. In particular, one never needs to α-rename bound names in order to avoid clashes with free variable names. Moreover, the implementation of capture-avoiding substitution is made significantly simpler by the separation of bound and free variables.

One drawback that remains about the locally nameless representation is that it is not strictly-speaking isomorphic to λ-terms. Here again, two terms may be α-equivalence but not syntactically equal. Even though many results can be formalized using the un-quotiented locally named representation, the construction of a quotient structure is required at some point for the sake of adequacy of the formalization.

2.3 The de Bruijn representation

There exists another standard approach to representing λ-terms. Using de Bruijn indices [1972], one can build a data type that is isomorphic to the set of λ-terms. In this representation, abstractions do not mention any name (they are "nameless"), and each variable bears a natural number that indicates the number of abstractions to be passed by before reaching the abstraction to which the variable is bound. More precisely, a variable marked with an index i points towards the "$(i+1)$-th" enclosing abstraction.

The grammar for terms in de Bruijn syntax, which appears next, includes variables built upon an index and nameless abstractions.

$$t \quad := \quad \mathsf{var}\, i \quad | \quad \mathsf{abs}\, t \quad | \quad \mathsf{app}\, t\, t$$

For example, the λ-term "$\lambda x.\, x$" is represented as "$\mathsf{abs}\,(\mathsf{var}\, 0)$", which can be also written "$\lambda.\, 0$". Similarly, the term "$\lambda x.\, ((\lambda y.\, y\, x)\, x)$" is represented as "$\lambda.\, ((\lambda.\, 0\, 1)\, 0)$".

The key advantage of using indices is that no quotient structure is required. Moreover, no α-renaming operation is ever needed when reasoning on λ-terms. However, the de Bruijn representation suffers from one major drawback: indices are very sensitive to changes in the term in which they occur. In particular, de Bruijn terms involve a *shifting* operation, which consists in incrementing in a term the value of all the indices that are greater than a given bound. It is often the case that conventional paper proofs need to undergo nontrivial arrangements in order to accommodate shifting. For example, in a proof of type soundness for a system with dependent types, the statement and proof of the weakening lemma is made significantly more complex because shifting needs to be applied to some of the values from the typing context.

The de Bruijn representation has shown its effectiveness in proofs of complex theorems, like Barras and Werner's formalization of Coq in Coq [1997]. Nevertheless, a number of researchers find the gap too large between the informal presentation and the formal de Bruijn presentation of a same theory [Aydemir et al., 2005].

2.4 The locally nameless representation

The locally nameless combines the benefits of the locally named representation with those of the de Bruijn representation. By using de Bruijn indices to represent bound variables, it avoids the introduction of α-equivalence classes. In the same time, by using names to represent free variables, it avoids the need for shifting de Bruijn indices.

The grammar of locally nameless terms thus involves a constructor for bound variables, built upon a de Bruijn index, and a constructor for free variables, built upon a name. Abstractions, like in de Bruijn syntax, are nameless: they do not carry any name.

$$t \quad := \quad \mathsf{bvar}\, i \quad | \quad \mathsf{fvar}\, x \quad | \quad \mathsf{abs}\, t \quad | \quad \mathsf{app}\, t\, t$$

For example, the λ-term "$\lambda x.\, x\, y$", which contains a bound variable x and a free variable y, is represented in locally nameless syntax as "$\mathsf{abs}\,(\mathsf{app}\,(\mathsf{bvar}\, 0)\,(\mathsf{fvar}\, y))$", which may also be written "$\lambda.\, 0\, y$". Note that not all syntactic terms correspond to an actual

λ-term. For instance, "$\mathsf{abs}\,(\mathsf{bvar}\,1)$" is not a valid locally nameless term because the bound variable with index 1 does not refer to any abstraction within its term. This issue of improper terms is addressed later on (§3.3).

Free variables are represented using names, also called *atoms*. Atoms can be implemented using any datatype that support a comparison function and a fresh name generator. The comparison function is used to test whether two atoms are equal (i.e., equality on atoms needs to be decidable). The fresh name generator, written "fresh", is used to pick an atom fresh from any given finite set of atoms (in particular, there should be infinitely many atoms). In practice, we usually implement atoms using natural numbers.

3 Operations on locally nameless terms

This section describes the operations used to manipulate locally nameless terms. In particular, two operations are central to this representation. *Variable opening* turns some bound variables into free variables. It is used to investigate the body of an abstraction. *Variable closing* turns some free variables into bound variables. It is used to build an abstraction given a representation of its body. In this section, we also explain how to rule out ill-formed terms allowed by the locally nameless syntax.

Note that the definitions and lemmas presented in this section are not very novel. Most of them have appeared either in Gordon's early work (1993) on locally nameless syntax or in McKinna and Pollack's work (1993) on the locally named representation, which has a lot in common with the locally nameless representation.

3.1 Variable opening

With the named representation, an abstraction takes the form "$\lambda x.\, t$". To investigate the body of this abstraction, we simply works with the term t. With the locally nameless representation, an abstraction has the form "$\mathsf{abs}\, t$" and it is our responsibility to provide a fresh name x to *open* the abstraction. The result of applying the *variable opening* operation to t and x is a term, written t^x, that describes the body of the abstraction "$\mathsf{abs}\, t$". More precisely, given an abstraction "$\mathsf{abs}\, t$" and a variable name x that does not appear in t, the term t^x is a copy of t in which all the bound variables referring to the outer abstraction of "$\mathsf{abs}\, t$" have been replaced with the free variable "$\mathsf{fvar}\, x$". For example, consider the abstraction "$\mathsf{abs}\,(\mathsf{app}\,(\mathsf{abs}\,(\mathsf{app}\,(\mathsf{bvar}\,0)\,(\mathsf{bvar}\,1)))\,(\mathsf{bvar}\,0))$"; the opening of its body with the name x is the term "$\mathsf{app}\,(\mathsf{abs}\,(\mathsf{app}\,(\mathsf{bvar}\,0)\,(\mathsf{fvar}\,x)))(\mathsf{fvar}\,x)$ ".

The implementation of variable opening needs to traverse a term recursively, and find all the leaves of the form "$\mathsf{bvar}\, i$" whose index i is equal to the number of abstractions enclosing that variable. Variable opening is thus defined in terms of a recursive function, written "$\{k \to x\}\, t$", that keeps track of the number k of abstractions that have been passed by. Initially, the value of k is 0, so variable opening is defined as:

$$t^x \quad \equiv \quad \{0 \to x\}\, t$$

The value of k is then incremented each time an abstraction is traversed. When reaching a bound variable with index i, the value of i is compared against the current value of k. If i is equal to k, then the bound variable is replaced with the free variable named x, otherwise it is unchanged. Note that free variables already occurring in the term are never affected by a variable opening operation.

$$\begin{array}{lcl}
\{k \to x\}\,(\mathsf{bvar}\,i) & \equiv & \text{if } (i = k) \text{ then } (\mathsf{fvar}\,x) \text{ else } (\mathsf{bvar}\,i) \\
\{k \to x\}\,(\mathsf{fvar}\,y) & \equiv & \mathsf{fvar}\,y \\
\{k \to x\}\,(\mathsf{app}\,t_1\,t_2) & \equiv & \mathsf{app}\,(\{k \to x\}\,t_1)\,(\{k \to x\}\,t_2) \\
\{k \to x\}\,(\mathsf{abs}\,t) & \equiv & \mathsf{abs}\,(\{(k+1) \to x\}\,t)
\end{array}$$

3.2 Variable closing

Symmetrically to variable opening, we may want to build an abstraction given its body. With the named representation, we consider a term t and a name x, and we simply build the abstraction "$\lambda x.\, t$". All the variables named x are abstracted, except those that already appear below an abstraction named x. With the locally nameless representation, we consider a term t and a name x to be abstracted in t, and we build a term, written $\backslash^x t$, by applying the *variable closing* operation to t and x. All the variables named x occurring in t are abstracted, without exception (indeed, no shadowing is possible with the locally nameless syntax). The abstraction may then be constructed as "$\mathsf{abs}\,(\backslash^x t)$". More precisely, the term $\backslash^x t$ is a copy of t in which all the free variables named x have been replaced with a bound variable. The indices of those variables are chosen in such a way that all the bound variables introduced are pointing towards the outer abstraction of "$\mathsf{abs}\,(\backslash^x t)$".

The implementation of variable closing follows a pattern similar to the implementation of variable opening. Its implementation is based on a recursive function, written "$\{k \leftarrow x\}\,t$", that keeps track of the number k of abstractions that have been passed by. Again, the value of k is 0 initially and it is incremented at each abstraction. Variable closing is defined as follows:

$$\backslash^x t \quad \equiv \quad \{0 \leftarrow x\}\,t$$

When the recursive function reaches a free variable with name y, it compares the name y with the name x. If the two names match, then the free variable y is replaced with a bound variable of index k, otherwise it is left unchanged. Note that bound variables already occurring in the term are never affected by variable closing.

$$\begin{array}{lcl}
\{k \leftarrow x\}\,(\mathsf{bvar}\,i) & \equiv & \mathsf{bvar}\,i \\
\{k \leftarrow x\}\,(\mathsf{fvar}\,y) & \equiv & \text{if } (x = y) \text{ then } (\mathsf{bvar}\,k) \text{ else } (\mathsf{fvar}\,y) \\
\{k \leftarrow x\}\,(\mathsf{app}\,t_1\,t_2) & \equiv & \mathsf{app}\,(\{k \leftarrow x\}\,t_1)\,(\{k \leftarrow x\}\,t_2) \\
\{k \leftarrow x\}\,(\mathsf{abs}\,t) & \equiv & \mathsf{abs}\,(\{(k+1) \leftarrow x\}\,t)
\end{array}$$

Variable closing is effectively the inverse function of the variable opening operation. Opening the body t of an abstraction with a fresh name x and then closing it with the same name x returns t. Symmetrically, closing a term t with a name x and then opening it with the same name x returns t. The corresponding formal statements, shown below, include technical side-conditions whose meaning is defined further on.

$$\begin{array}{lll}
\textsc{close\_open\_var}: & \backslash^x(t^x) = t & \text{when } \; x \# t \\
\textsc{open\_close\_var}: & (\backslash^x t)^x = t & \text{when } \; \mathsf{lc}\,t
\end{array}$$

As a corollary, both "variable opening with a fresh name" and "variable closing" are injective operations on the set of well-formed locally nameless terms. For example, injectivity of variable opening is useful to prove that two abstractions "$\mathsf{abs}\, t_1$" and "$\mathsf{abs}\, t_2$" are equal from the knowledge that their bodies opened with the same fresh name x are equal, i.e., from the fact that ${t_1}^x$ is equal to ${t_2}^x$ (see §6.7).

3.3 Locally-closed terms

As suggested in the previous section, the locally nameless syntax contains objects that do not correspond to any valid λ-term. For instance, "$\mathsf{abs}\, 3$" is such an improper syntactic object, since the bound variable with index 3 does not refer to any abstraction inside the term. We need to ensure that terms do not contain any such *dangling bound variable*. We say of well-formed terms that they are *locally closed*. The purpose of this section is to give a formal characterization of the set of locally closed terms.

Two approaches are possible. The first one consists in investigating the term recursively, opening every abstraction with a name, and checking that no bound variable is ever reached. The second possible approach relies on an analysis of bound variables, for checking that each bound variable has an index smaller than the number of enclosing abstractions. We start by describing the first approach, which is the most helpful for formally reasoning on terms represented in locally nameless style, and study the approach based on indices afterwards.

The *local closure* predicate, written "$\mathsf{lc}\, t$", characterizes terms that are locally closed. It is defined using three inductive rules. The first one states that any free variable is locally closed. The second one states that an application is locally closed if its two branches are locally closed. The third and last one states that an abstraction is locally closed if its body opened with some name is itself locally closed. Notice that a bound variable on its own is never locally closed.

$$\frac{}{\mathsf{lc}\,(\mathsf{fvar}\, x)}\ \textsc{lc-var'} \qquad \frac{\mathsf{lc}\, t_1 \qquad \mathsf{lc}\, t_2}{\mathsf{lc}\,(t_1\, t_2)}\ \textsc{lc-app'} \qquad \frac{\mathsf{lc}\,(t^x)}{\mathsf{lc}\,(\mathsf{abs}\, t)}\ \textsc{lc-abs'}$$

In practice, we use a slightly different rule to deal with abstractions. In the rule LC-VAR', the premise $\mathsf{lc}\,(t^x)$ is required to hold for one single name x. Instead, we are going to require $\mathsf{lc}\,(t^x)$ to hold for cofinitely-many names x. More precisely, we consider that an abstraction "$\mathsf{abs}\, t$" is locally closed if there exists a finite set of names L such that, for any name x not in L, the term t^x is locally closed.

$$\frac{}{\mathsf{lc}\,(\mathsf{fvar}\, x)}\ \textsc{lc-var} \qquad \frac{\mathsf{lc}\, t_1 \qquad \mathsf{lc}\, t_2}{\mathsf{lc}\,(t_1\, t_2)}\ \textsc{lc-app} \qquad \frac{\forall\, x \notin L, \quad \mathsf{lc}\,(t^x)}{\mathsf{lc}\,(\mathsf{abs}\, t)}\ \textsc{lc-abs}$$

The motivation for the cofinite quantification will be discussed in details later on (§4.2).

Another way of characterizing locally closed terms is based on the analysis of the value of indices appearing in terms. Intuitively, a term is locally closed if and only if all its bound variables have an index small enough that they actually point to an abstraction inside the term. To ensure that this is the case, it suffices to verify that every bound variable has an index smaller than the number of enclosing abstractions.

This intuition is formalized through the predicate "*t is closed at level k*", written "$\mathsf{lc\_at}\, k\, t$". This predicate is defined recursively on the structure of the term t. The parameter k is used to maintain the current depth. A bound variable is closed at level

k if and only if its index is smaller than k. A free variable is closed at any level. The complete definition of the predicate "*closed at level k*" appears below.

$$\begin{array}{lcl}
\mathsf{lc\_at}\,k\,(\mathsf{bvar}\,i) & \equiv & i < k \\
\mathsf{lc\_at}\,k\,(\mathsf{fvar}\,x) & \equiv & \mathsf{True} \\
\mathsf{lc\_at}\,k\,(\mathsf{app}\,t_1\,t_2) & \equiv & \mathsf{lc\_at}\,k\,t_1 \,\wedge\, \mathsf{lc\_at}\,k\,t_2 \\
\mathsf{lc\_at}\,k\,(\mathsf{abs}\,t) & \equiv & \mathsf{lc\_at}\,(k+1)\,t
\end{array}$$

It can be proved that a term is locally closed if and only if it is closed at level 0.

$$\textsc{lc\_from\_lc\_at}: \quad \mathsf{lc}\,t \iff \mathsf{lc\_at}\,0\,t$$

In conclusion, there are two approaches to defining local closure. Throughout the rest of the paper, we only use the inductive definition. Indeed, it involves simpler rules that do not involve an auxiliary variable k for describing the current depth. Moreover, the inductive definition gives rise to an induction principle that matches more closely the way inductions on λ-terms are performed in informal proofs.

3.4 Restriction to locally-closed terms

When formally reasoning on locally nameless terms, we want to manipulate only locally closed terms. Indeed, in general, it does not make sense to state properties on syntactic objects that do not correspond to any λ-term. Thus, we need to ensure that any function that manipulates terms preserves the local closure property, and that any relation defined on terms is restricted to locally closed terms. The explanation of how to implement this restriction is postponed to the second part of the paper (see §4.5). Here, we only describe the properties of the basic operations on terms with respect to local closure.

One auxiliary definition is useful for stating local closure properties. The predicate "$\mathsf{body}\,t$" asserts that t describes the body of a locally closed abstraction. Its definition is equivalent to the premise of the rule LC-ABS that defines locally closed abstractions.

$$\mathsf{body}\,t \quad \equiv \quad \exists\, L,\ \forall\, x \notin L,\ \mathsf{lc}\,(t^x)$$

An abstraction is locally closed if and only if its body satisfies the predicate body:

$$\textsc{lc\_abs\_iff\_body}: \quad \mathsf{lc}\,(\mathsf{abs}\,t) \iff \mathsf{body}\,t$$

The definition of body helps stating lemmas describing the behaviour of variable opening and variable closing operation with respect to local closure. First, if t is a body, then t opened with variable x is locally closed. Second, if t is locally closed, then the closing of t with respect to a variable x yields a valid body.

$$\begin{array}{lll}
\textsc{open\_var\_lc}: & \mathsf{body}\,t & \Rightarrow\ \mathsf{lc}\,(t^x) \\
\textsc{close\_var\_lc}: & \mathsf{lc}\,t & \Rightarrow\ \mathsf{body}\,({}^{\backslash x}t)
\end{array}$$

3.5 Free variables and substitution

The free variable function and the substitution function are essential to reasoning on λ-terms. In what follows, we describe the definition and properties of these two operations on locally nameless syntax.

Since free variables are syntactically distinguished from bound variables, the computation of the set of free variable names "fv(t)" occurring in a term t is totally straightforward: it suffices to gather all the names that occur in t.

$$\begin{array}{lcl}
\mathrm{fv}(\mathsf{bvar}\,i) & \equiv & \emptyset \\
\mathrm{fv}(\mathsf{fvar}\,x) & \equiv & \{x\} \\
\mathrm{fv}(\mathsf{app}\,t_1\,t_2) & \equiv & \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2) \\
\mathrm{fv}(\mathsf{abs}\,t) & \equiv & \mathrm{fv}(t)
\end{array}$$

Throughout the paper, a name x is said to be *fresh* for a term t, written "$x \# t$", if x does not belong to the set of free variables of t. Moreover, a term t is said to be *closed* if it has no free variables at all.

$$\begin{array}{lcl}
x \# t & \equiv & (x \notin \mathrm{fv}(t)) \\
\mathsf{closed}\,t & \equiv & (\mathrm{fv}(t) = \emptyset)
\end{array}$$

There are two properties of the free variable function that are specific to the locally nameless representation. They describe the interaction of fv with variable opening and variable closing. First, opening a body t with a name x potentially adds x to the set of its free variables. Second, closing a term t with respect to a name x removes x from the set of its free variables. These results, which are useful to reason on freshness, can be formally stated as follows.

$$\begin{array}{lll}
\textsc{open\_var\_fv}: & \mathrm{fv}(t^x) & \subseteq \mathrm{fv}(t) \cup \{x\} \\
\textsc{close\_var\_fv}: & \mathrm{fv}({}^{\backslash x}t) & = \mathrm{fv}(t) \setminus \{x\}
\end{array}$$

Another key operation is the substitution function. The notation "$[x \to u]\,t$" describes a copy of the term t in which all occurrences of x have been replaced with the term u. we can implement the substitution with a recursive function that follows the structure of the term t. When reaching a free variable named y, the function simply compares y with x, and, in case the two names are equal, it replaces the free variable y with the term u. The complete description follows. Observe that we need not worry about shadowing nor variable capture.

$$\begin{array}{lcl}
[x \to u]\,(\mathsf{bvar}\,i) & \equiv & \mathsf{bvar}\,i \\
[x \to u]\,(\mathsf{fvar}\,y) & \equiv & \text{if } (x = y) \text{ then } u \text{ else } (\mathsf{fvar}\,y) \\
[x \to u]\,(\mathsf{app}\,t_1\,t_2) & \equiv & \mathsf{app}\,([x \to u]\,t_1)\,([x \to u]\,t_2) \\
[x \to u]\,(\mathsf{abs}\,t) & \equiv & \mathsf{abs}\,([x \to u]\,t)
\end{array}$$

There are several properties of the substitution function that are specific to the locally nameless representation. First, substitution preserves the local closure property.

$$\begin{array}{llcccl}
\textsc{subst\_lc}: & \mathsf{lc}\,t & \wedge & \mathsf{lc}\,u & \Rightarrow & \mathsf{lc}\,([x \to u]\,t) \\
\textsc{subst\_body}: & \mathsf{body}\,t & \wedge & \mathsf{lc}\,u & \Rightarrow & \mathsf{body}\,([x \to u]\,t)
\end{array}$$

Second, substitution commutes with variable opening and variable closing, given suitable freshness conditions. Those two results are key for establishing the preservation of a given property on terms through substitution (see, e.g., the proof that substitution preserves typing, §6.2).

$$\begin{array}{lll} \textsc{subst\_open\_var}: & [x \to u]\,(t^y) \;=\; ([x \to u]\,t)^y & \text{when } \; x \neq y \wedge \mathsf{lc}\,u \\ \textsc{subst\_close\_var}: & [x \to u]\,(\backslash^y t) \;=\; \backslash^y([x \to u]\,t) & \text{when } \; x \neq y \wedge y \# u \end{array}$$

Other standard properties of the substitution function can be easily derived. For instance, the substitution for a fresh name behaves as the identity function.

$$\textsc{subst\_fresh}: \qquad x \# t \quad \Rightarrow \quad [x \to u]\,t \;=\; t$$

The definition of substitution presented above can be generalized so as to support multi-substitutions, where several names are substituted in the same time. Such a multi-substitution function is parameterized by a map from variable names to terms. When reaching a free variable whose name belongs to the domain of that map, the function replaces it with the term bound to that name in the map.

3.6 β-reduction and opening

Beta-reduction is a fundamental operation on λ-terms. We first show how β-reduction can be implemented in terms of the substitution function and then explain that it can be implemented more directly in terms of a generalization of the variable opening operation.

When working with a named representation, the β-reduction rule is stated as:

$$((\lambda x.\,t)\,u) \quad \longrightarrow_\beta \quad [x \to u]\,t$$

With the locally nameless representation, a β-redex takes the form "$\mathsf{app}\,(\mathsf{abs}\,t)\,u$". One could implement β-reduction by first opening the body t of the abstraction with a fresh name x, obtaining the term t^x, and then substituting u for x in that term. Thus, the β-reduction rule can be stated as follows:

$$\mathsf{app}\,(\mathsf{abs}\,t)\,u \quad \longrightarrow_\beta \quad [x \to u]\,(t^x) \qquad \text{for any } x \# t$$

The above statement describes a correct and usable definition, yet a more direct definition can be devised. Let us analyse the computations described by the expression "$[x \to u]\,(t^x)$". It consists in replacing all the bound variables in t that point to the outer abstraction of "$\mathsf{abs}\,t$" with a fresh free variable x, and then replacing all occurrences of x with the term u. This is equivalent to directly replacing all the relevant bound variables in t by u, thereby avoiding the introduction of a temporary name x.

This suggests a new operation that generalizes variable opening in the following way: instead of replacing relevant bound variables with a free variable, it replaces those bound variables with an arbitrary given term. This new operation, which we call *opening*, allows to β-reduce an abstraction "$\mathsf{abs}\,t$" onto a term u. As it is strictly more general than variable opening, we reuse the same notation, and write t^u. The new statement of the β-reduction rule, which no longer requires the introduction of an arbitrary fresh name, appears below.

$$\mathsf{app}\,(\mathsf{abs}\,t)\,u \quad \longrightarrow_\beta \quad t^u$$

The implementation of opening differs from the implementation of variable opening only on the case for bound variables. Opening is defined in terms of an auxiliary recursive function, written "$\{k \to u\}\, t$", that describes the fact that bound variables at depth k are to be replaced by the term u inside the term t.

$$t^{u} \quad \equiv \quad \{0 \to u\}\, t$$

$$\begin{array}{lcl} \{k \to u\}\, (\mathsf{bvar}\, i) & \equiv & \text{if } (i = k) \text{ then } u \text{ else } (\mathsf{bvar}\, i) \\ \{k \to u\}\, (\mathsf{fvar}\, y) & \equiv & \mathsf{fvar}\, y \\ \{k \to u\}\, (\mathsf{app}\, t_1\, t_2) & \equiv & \mathsf{app}\, (\{k \to u\}\, t_1)\, (\{k \to u\}\, t_2) \\ \{k \to u\}\, (\mathsf{abs}\, t) & \equiv & \mathsf{abs}\, (\{(k+1) \to u\}\, t) \end{array}$$

We can prove that the opening operation preserves local closure. (This result generalizes the lemma OPEN_VAR_LC.)

$$\text{OPEN\_LC:} \qquad \mathsf{body}\, t \quad \wedge \quad \mathsf{lc}\, u \quad \Rightarrow \quad \mathsf{lc}\, (t^{u})$$

Variable opening can be recovered as a particular instance of opening. Indeed, variable opening with a name x is the same as opening with a free variable named x. Thus, one may define variable opening in terms of opening, and save the need to define both operations independently. Formally:

$$t^{x} \quad \equiv \quad t^{(\mathsf{fvar}\, x)}$$

3.7 Connections between substitution and opening

Substitution replaces free variables with terms, while variable closing replaces free variables with bound variables and opening replaces bound variables with terms. Thus, there exists strong connections relating these three functions. The purpose of the following investigation is to establish these connections, explain why they hold, and suggest when they need to be exploited in reasoning.

As explained in the paragraph that motivates the introduction of the open function, opening with a term u is the same as opening with a fresh variable x and then substituting u for x. This relationship provides a way to decompose an opening operation in terms of a variable opening operation and a substitution operation.

$$\text{SUBST\_INTRO:} \qquad t^{u} \;=\; [x \to u]\, (t^{x}) \quad \text{when } x \;\#\; t$$

This property is key to proving properties of β-reduction in terms of a corresponding property about substitution. For instance, the fact that β-reduction preserves typing is proved using the fact that substitution preserves typing (see §6.3).

The property SUBST_INTRO is intuitively a consequence of a more general result describing the distributivity of substitution over open:

$$\text{SUBST\_OPEN:} \qquad [x \to u]\, (t^{v}) \;=\; ([x \to u]\, t)^{([x \to u]\, v)} \quad \text{when } \mathsf{lc}\, u$$

This lemma describes how a substitution commutes with opening. It is involved in the proof that two independent β-reductions can be permuted, a result used to establish the confluence of β-reduction (see §6.8).

In a similar way as SUBST_INTRO relates substitution and opening, there exists a relation between substitution and variable closing. It states than closing with respect to a variable named x is equivalent to first renaming all occurrences of x into y and then closing with respect to y, for any fresh name y. Here and thereafter, we write "$[x \to y]$" as a shorthand for the renaming operation "$[x \to \mathsf{fvar}\, y]$".

$$\text{CLOSE\_VAR\_RENAME:} \qquad {}^{\backslash x}t \;=\; {}^{\backslash y}([x \to y]\, t) \quad \text{when } y \# t$$

This lemma is useful for establishing that the result of a function defined recursively on λ-terms does not depend on the fresh names being chosen for investigating bodies of abstractions (see §6.9).

A corollary of the lemma SUBST_INTRO is that substitution can be defined in terms of opening and variable closing. More precisely, in order to replace all occurrences of a variable x with a term u inside a term t, it suffices to close t with respect to x, and then open the resulting term with u. This amounts to replacing all occurrences of the free variable x with bound variables, and then replacing all these freshly introduced bound variables with copies of the term u.

$$\text{SUBST\_AS\_CLOSE\_OPEN:} \qquad [x \to u]\, t \;=\; \left({}^{\backslash x}t\right)^{u}$$

In fact, this property can be used as an elegant definition of the substitution function. Defining substitution in terms of opening and variable closing helps reduce the number of recursive definitions involved when programming with locally nameless syntax. However, in the context of reasoning, a direct recursive function turns out to be more convenient, as it avoids the burden of stating and exploiting lemmas describing how the substitution function distributes over constructors from the syntax of terms.

3.8 Proofs

Most of the lemmas presented so far have relatively simple proofs, that can be formalized in a proof assistant in just a few lines. Proofs fall in three categories.

Firstly, a number of low-level properties are proved by induction on the structure of a term. Lemmas CLOSE_OPEN_VAR, OPEN_VAR_FV, CLOSE_VAR_FV, SUBST_FRESH, SUBST_OPEN and CLOSE_VAR_RENAME are proved in this way. For example, consider the lemma SUBST_OPEN. Given a locally closed term u, we prove by induction on the structure of t that, for any index k, the following statement holds:

$$[x \to u]\, (\{k \to v\}\, t) \;=\; \{k \to ([x \to u]\, v)\}\, ([x \to u]\, t)$$

All cases are easy except one, where a lemma called OPEN_REC_LC needs to be exploited. This lemma, stated below, asserts that substitution for a de Bruijn index does not affect a locally closed term.

$$\text{OPEN\_REC\_LC:} \qquad \mathsf{lc}\, u \quad \Rightarrow \quad \forall k. \quad (\{k \to v\}\, u) = u$$

Secondly, a number of lemmas are proved by induction on the derivation of local closure of a term. Lemmas OPEN_CLOSE_VAR, CLOSE_VAR_LC, SUBST_LC and OPEN_REC_LC are proved this way. The proofs of SUBST_LC is straightforward, however the proof of the other lemmas involve a technical intermediate result to handle the abstraction case. The proof of OPEN_REC_LC exploits the following lemma (with j equal to 0):

$$i \neq j \quad \wedge \quad \{i \to u\}\, (\{j \to v\}\, t) \;=\; \{i \to u\}\, t \quad \Rightarrow \quad (\{i \to u\}\, t) = t$$

Similarly, the proof of OPEN_CLOSE_VAR and that of CLOSE_VAR_LC exploit the fact:

$$\{i \to y\}\,\{j \to z\}\,\{j \leftarrow x\}\,t \;=\; \{j \to z\}\,\{j \leftarrow x\}\,\{i \to y\}\,t \quad \text{when } i \neq j \wedge x \neq y \wedge y \,\#\, t$$

Those two intermediate results admit direct proofs by induction on the structure of t.

In the previous paragraph, we have explained how to prove results by induction on the derivation of a local closure judgment. There is another possible way of conducting those proofs, based on the alternative local closure predicate "$\mathsf{lc\_at}\,k\,t$". Because the level k is explicitly exposed with this judgment, the induction can be conducted on the structure of t. The resulting proofs are slightly simpler. In particular, they avoid the two technical intermediate lemmas described in the previous paragraph. That said, this alternative proof technique comes with a significant entry cost. Indeed, it requires us to define the function $\mathsf{lc\_at}$ and to prove the predicate "$\mathsf{lc\_at}\,0$" equivalent to lc. In practice, we have found that the cost of defining the function $\mathsf{lc\_at}$, which has a size linear in the number of constructions of the language, is greater than the cost of proving two intermediate lemmas, whose statements and proofs are just a few lines long. For this reason, we prefer conducting our proofs by induction on the inductively-defined local closure predicate.

Finally, several properties are deducible from other lemmas. First, SUBST_INTRO can be derived from SUBST_OPEN by instantiating v as "$\mathsf{fvar}\,x$" and exploiting the lemma SUBST_FRESH to show that "$[x \to u]\,t$" is equal to t since x is fresh for t.[1] Second, the lemma SUBST_BODY is a corrolary of SUBST_LC (using SUBST_OPEN_VAR). Finally, the lemma OPEN_LC can be deduced from SUBST_INTRO and SUBST_LC: to prove t^u locally closed, one first rewrite this term as $[x \to u]\,(t^x)$ and then invoke the fact that substitution preserves local closure.

3.9 Summary

The infrastructure associated with the locally nameless representation can be set up as follows:

1. Define the syntax in locally nameless style, that is, with distinct constructors for bound and free variables, and with nameless abstractions.
2. Define the opening and the variable closing operations. Derive the definition of variable opening and of β-reduction from the definition of opening.
3. Define the free variables function and the substitution function. Define the local closure predicate and its auxiliary "body" predicate.
4. State and prove the properties of the operations on terms that are needed in the development to be carried out.

4 Formal definitions in locally nameless style

In this section, we explain and illustrate how to formally state inductive definitions on λ-terms in the locally nameless representation. Starting from a definition in the named representation, three steps are involved for reaching a correct and practical locally

[1] Proving SUBST_INTRO from SUBST_OPEN requires an assumption about the local closure of the term being substituted in, although the lemma SUBST_INTRO technically holds even without this side condition.

nameless definition. The first step is to replace named abstractions with nameless abstractions, and use variable opening to open bodies of abstractions. The second step is to quantify properly the names introduced for variable opening. For this purpose, we use a particular technique based on the cofinite quantification of names [Aydemir et al., 2008]. Other quantifications are possible, but the cofinite quantification offers key advantages from an engineering point of view. The details of the motivation for this technique and the justification of its correctness are out of the scope of this paper. This paper only contains a short introduction to the cofinite quantification technique. The third and last step consists in adding a number of premises to inductive rules so as to ensure that inductive judgments are restricted to locally closed terms.

4.1 Introduction of variable opening in inductive rules

Consider the standard the typing rule for abstraction in the simply-typed λ-calculus, shown below on the left-hand side. To obtain the locally nameless version of that rule we need to turn the named abstraction "$\lambda x.\, t$" into a nameless abstraction "$\mathsf{abs}\, t$" and use an explicit variable opening operation to build the term t^x, which describes the body of that abstraction. We obtain the rule shown below on the right-hand side.

$$\frac{E,\ x : T_1 \vdash t : T_2}{E \vdash \lambda x.\, t : T_1 \to T_2}\ \text{TYPING-ABS-WITH-NAMES} \qquad \frac{E,\ x : T_1 \vdash t^x : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2}\ \text{TYPING-ABS-LOCALLY-NAMELESS}$$

The transformation from the named version to the locally nameless version of an inductive rule is very systematic. In the next section (§5), we will see many examples of such transformations.

4.2 Quantification of free variable names

The rule TYPING-ABS-LOCALLY-NAMELESS is not explicit about the freshness side-conditions that the variable name x should satisfy. Since we use x to open the body t of the abstraction, the name x should be fresh from t. Moreover, since we extend the environment E with a binding for x, the name x should be fresh from the domain of E.

If we explicitly include the freshness condition $x \notin \mathrm{fv}(t) \cup \mathrm{dom}(E)$, we get the rule TYPING-ABS-EXISTENTIAL, shown next. In this rule, the name x is existential quantified: it suffices to exhibit a typing derivation of t^x for one fresh name x in order to build a typing derivation for "$\mathsf{abs}\, t$". However, we could also require that t^x admits the type T_2 for any fresh name x. In this case, we obtain the rule TYPING-ABS-UNIVERSAL.

$$\frac{x \notin \mathrm{fv}(t) \cup \mathrm{dom}(E) \qquad E,\ x : T_1 \vdash t^x : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2}\ \text{TYPING-ABS-EXISTENTIAL}$$

$$\frac{\forall x \notin \mathrm{fv}(t) \cup \mathrm{dom}(E), \qquad E,\ x : T_1 \vdash t^x : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2}\ \text{TYPING-ABS-UNIVERSAL}$$

We advocate using a third rule, based on a cofinite quantification. The premise of this rule, shown next, requires the existence of a finite set of names, called L, such that the term t^x admits the type T_2 for any name x that does not belong to the set L.

$$\frac{\forall x \notin L, \qquad E,\, x : T_1 \vdash t^x : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2}\ \textsc{typing-abs-cofinite}$$

One advantage of the cofinitely-quantified typing rule is that we do not need to work out what x should be fresh from. Indeed, if t^x admits the type T_2 for a cofinite number of names, we can certainly find at least one fresh name such that t^x admits the type T_2.

The existential rule is very convenient as an introduction form: to build a typing derivation for an abstraction, it suffices to type-check its body for one fresh name. However, this rule is very weak as an elimination form: given the assumption that an abstraction is well-typed, we only learn that its body is well-typed for one particular fresh name. On the contrary, the universal rule is very convenient as an elimination form: if we have a well-typed abstraction $\mathsf{abs}\, t$, we can immediately obtain the knowledge that t^x is well-typed for any fresh name x. Yet, the universal rule is hard to use as an introduction rule: it requires us to prove that t^x for every possible fresh name x.

The cofinite rule is a compromise between the existential rule and the cofinite rule. As an elimination form, the cofinite rule is nearly as strong as the universal rule: it gives us knowledge that t^x is well-typed for infinitely many names. As an introduction form, the cofinite rule is not as bad as the universal rule. The cofinite quantification gives us some slack, in the sense that we are able to exclude from the quantification an arbitrary finite set of names. We will give in §6 examples where the ability to exclude particular names is crucial.

4.3 Introduction lemma

While the cofinite rule is much better than the universal rule as an introduction form, it is not quite as powerful as the existential rule. Indeed, the cofinite rule still requires us to establish a result for infinitely many names before we can apply it. There are cases where the cofinite rule is not good enough as an introduction form in the sense that we are only able to build a proof of the premise for one fresh name, and not for infinitely many names. (An example will be given in §6.7.)

In such situation, we need to resort to an *introduction lemma*, which simply states that the existential rule is admissible. For the typing judgment, the introduction lemma states that it suffices to show that t^x is well-typed for one fresh name x in order to deduce that $\mathsf{abs}\, t$ is well-typed.

$$\textsc{typing-abs-intro:} \qquad \begin{cases} x \notin \mathrm{fv}(t) \cup \mathrm{dom}(E) \\ E,\, x : T_1 \vdash t^x : T_2 \end{cases} \quad \Rightarrow \quad E \vdash \mathsf{abs}\, t : T_1 \to T_2$$

The proof of this introduction lemma is based on a *renaming lemma*. The renaming lemma states that if t^x is well-typed for one fresh name x then t^y is also well-typed for any other fresh name y. Intuitively, renaming lemmas capture the idea that the choice of names for free variable is irrelevant as long as the names are sufficiently fresh.

$$\textsc{typing\_rename:} \qquad \begin{cases} E,\, x : T_1 \vdash t^x : T_2 \\ x \notin \mathrm{fv}(t) \cup \mathrm{dom}(E) \\ y \notin \mathrm{fv}(t) \cup \mathrm{dom}(E) \end{cases} \quad \Rightarrow \quad E,\, y : T_1 \vdash t^y : T_2$$

The proof of an introduction lemma and of a renaming lemma will be given in §6.6.

4.4 Induction principle

When performing an induction over the an inductively-defined judgment involving cofinite quantification, the induction hypothesis provided in the abstraction case is quantified cofinitely. For example, consider the induction principle associated with the local closure judgment.

$$\textsc{induction\_principle\_for\_terms}:$$
$$\forall\, t,\quad \begin{cases} \forall\, x,\ P(\mathsf{fvar}\, x) \\ \forall\, t_1\; t_2,\ \ P(t_1) \Rightarrow P(t_2) \Rightarrow P(\mathsf{app}\, t_1\, t_2) \\ \forall\, L,\ \ (\forall\, x \notin L,\ \ P(t^x)) \Rightarrow P(\mathsf{abs}\, t) \end{cases} \quad\Rightarrow\quad (\forall\, t,\, \mathsf{lc}\, t \Rightarrow P(t))$$

When conducting a proof by induction, in the abstraction case we are given a finite set of names L and we are given an hypothesis about t^x for any x not in L. The induction hypothesis is very strong: it states that $P(t^x)$ holds for all cofinitely-many names x. In practice, the form of elimination associated with a cofinitely-quantified inductive rule appears to always be sufficiently strong, in the sense that there is no need to resort to the induction principle associated with the corresponding universally-quantified inductive rule.

4.5 Restriction to locally closed terms

In §3.4, we explained that all judgments on terms need to be restricted to locally closed terms. In this section, we show how this restriction can be implemented through the addition of extra premises in inductive rules. We start by illustrating this mechanism on an example, and then state a general construction rule.

Consider the definition of full β-reduction on λ-terms, written $t \longrightarrow t'$. It involves four rules: one rule for contracting a head β-redex, plus three rules for reducing under all possible evaluation contexts.

$$\frac{\mathsf{body}\, t \qquad \mathsf{lc}\, u}{\mathsf{app}\,(\mathsf{abs}\, t)\, u \longrightarrow t^u}\ \textsc{beta-reduce} \qquad\qquad \frac{t_1 \longrightarrow t_1' \qquad \mathsf{lc}\, t_2}{\mathsf{app}\, t_1\, t_2 \longrightarrow \mathsf{app}\, t_1'\, t_2}\ \textsc{beta-app-1}$$

$$\frac{\mathsf{lc}\, t_1 \qquad t_2 \longrightarrow t_2'}{\mathsf{app}\, t_1\, t_2 \longrightarrow \mathsf{app}\, t_1\, t_2'}\ \textsc{beta-app-2} \qquad\qquad \frac{\forall\, x \notin L,\quad t^x \longrightarrow {t'}^x}{\mathsf{abs}\, t \longrightarrow \mathsf{abs}\, t'}\ \textsc{beta-abs}$$

These rules include premises for ensuring that whenever the proposition $t \longrightarrow t'$ holds, both t and t' are locally closed. For instance, the rule BETA-REDUCE includes two such premises: one premise for ensuring that the argument u of the application is a locally closed term, written "$\mathsf{lc}\, u$", and another premise for ensuring that the abstraction involved is also locally closed, written "$\mathsf{body}\, t$". For the latter, we could have written "$\mathsf{lc}\,(\mathsf{abs}\, t)$", but the equivalent proposition "$\mathsf{body}\, t$" is both lighter and handier from a proof engineering point-of-view.

Not all terms involved in inductive definitions require a specific premise. For instance, the rule BETA-APP-1 does not require a premise stating the local closure of t_1. Indeed, the premise describing the reduction of this term, namely $t_1 \longrightarrow t_1'$, suffices to guarantee that t_1 is locally closed. Sometimes, no extra premise is needed at all, as it is the case for example in the rule BETA-ABS.

We can formally state and prove that whenever t reduces to t', both t and t' are locally closed. This is the matter of the following *regularity lemma*, whose proof is straightforward by induction on the definition of the β-reduction relation.

$$\textsc{beta\_regular}: \qquad t \longrightarrow t' \quad \Rightarrow \quad \mathsf{lc}\, t \quad \wedge \quad \mathsf{lc}\, t'$$

In general, given an inductive rule, a local closure hypothesis is required for each meta-variable describing a term that appears in the conclusion of the rule but not in any premise able to guarantee the local closure of this meta-variable. In most cases, as soon as the meta-variable is mentioned in at least one premise, it does not require an explicit local closure hypothesis.

Furthermore, one needs to ensure that all data-structures containing locally nameless terms do enforce local closure on these terms. For example, consider the formalization of the semantics of an imperative language with a store mapping locations to terms. In such a development, any time a store meta-variable is involved, one must be able to prove that this store contains only locally closed terms. A similar need occurs when formalizing languages whose types contain binders: typing contexts must be restricted to contain only types with locally closed representations (see, e.g., §5.4).

Fortunately, in practice, the number of extra premises needed for ensuring *regularity* generally remains quite small. Moreover, one can usually set up proof automation so that all the corresponding side-conditions can be discharged automatically when applying an inductive rule [Charguéraud, 2009]. Thus, the overall overhead associated with the need to enforce local closure throughout formalizations appears to be fairly reasonable in practice.

5 Examples of definitions in locally nameless style

We now present a series of examples of inductive definition in locally nameless style with cofinite quantification. The first purpose is to illustrate further the use of cofinite quantification as well as the addition of local closure premises. The second purpose is to define the judgments involved in the next section, where we focus on formal reasoning on locally nameless definitions. We consider the following examples: call-by-value β-reduction, parallel reduction, reflexive-transitive closure of β-reduction, big-step reduction, typing judgment in simply-typed λ-calculus, and typing and subtyping judgments in System $F_{<:}$.[2]

5.1 Small-step reductions on λ-terms

Our first example consists in the definition of call-by-value reduction on λ-terms. First, we define a predicate "value" in order to characterize values. In the pure λ-calculus, only locally closed abstraction are values, as stated by the rule VALUE-ABS. The definition of the call-by-value reduction predicate, written $t \longrightarrow_{\mathsf{cbv}} t'$, is defined by three rules: one rule to β-reduce the application of an abstraction to a value, one rule to reduce

[2] This paper does not describe the formalization of languages featuring mutable stores or exceptions, as these features are of limited interest with respect to binding issues. Details on the representation of such features can be found in the formal developments that the author has carried out [Charguéraud, 2009].

the right-hand side of an application, and one rule to reduce the left-hand side of an application when the right-hand side has already reduced to a value.

$$\frac{\mathsf{body}\, t}{\mathsf{value}\ (\mathsf{abs}\, t)}\ \textsc{value-abs} \qquad \frac{\mathsf{body}\, t \qquad \mathsf{value}\ u}{\mathsf{app}\ (\mathsf{abs}\, t)\ u\ \longrightarrow_{\mathsf{cbv}}\ t^{u}}\ \textsc{cbv-reduce}$$

$$\frac{t_1\ \longrightarrow_{\mathsf{cbv}}\ t_1' \qquad \mathsf{lc}\, t_2}{\mathsf{app}\, t_1\, t_2\ \longrightarrow_{\mathsf{cbv}}\ \mathsf{app}\, t_1'\, t_2}\ \textsc{cbv-app-1} \qquad \frac{\mathsf{value}\ t_1 \qquad t_2\ \longrightarrow_{\mathsf{cbv}}\ t_2'}{\mathsf{app}\, t_1\, t_2\ \longrightarrow_{\mathsf{cbv}}\ \mathsf{app}\, t_1\, t_2'}\ \textsc{cbv-app-2}$$

Observe that the rule CBV-APP-2 does not require any local closure premise because the property "$\mathsf{lc}\, t_1$" is implied by the assumption that t_1 is a value.

Another interesting variant of β-reduction is the parallel reduction predicate, which we will use to prove confluence of β-reduction. With the parallel reduction judgment, written $t\ \twoheadrightarrow\ t'$, both branches of an application can be reduced in parallel. Moreover, both the abstraction and the argument of a β-redex can be reduced before the redex is contracted. The formal rules are shown next. Notice that no local closure premise is needed to ensure the regularity of the parallel reduction relation.

$$\frac{}{\mathsf{fvar}\, x\ \twoheadrightarrow\ \mathsf{fvar}\, x}\ \textsc{para-var} \qquad \frac{\big(\forall x \notin L,\quad {t_1}^{x}\ \twoheadrightarrow\ {t_1'}^{x}\big) \qquad t_2\ \twoheadrightarrow\ t_2'}{\mathsf{app}\ (\mathsf{abs}\, t_1)\, t_2\ \twoheadrightarrow\ {t_1'}^{t_2'}}\ \textsc{para-reduce}$$

$$\frac{\forall x \notin L,\quad t^{x}\ \twoheadrightarrow\ {t'}^{x}}{\mathsf{abs}\, t\ \twoheadrightarrow\ \mathsf{abs}\, t'}\ \textsc{para-abs} \qquad \frac{t_1\ \twoheadrightarrow\ t_1' \qquad t_2\ \twoheadrightarrow\ t_2'}{\mathsf{app}\, t_1\, t_2\ \twoheadrightarrow\ \mathsf{app}\, t_1'\, t_2'}\ \textsc{para-app}$$

5.2 Multiple-step reductions on λ-terms

The reflexive-transitive closure of the β-reduction relation, written $t \longrightarrow^{*} t'$, can be defined using two rules: one rule for the empty reduction sequence, and one rule decomposing a non-empty reduction sequence by isolating its first reduction step. As any other relation on terms, we need to restrict $\longrightarrow^{*}$ to locally closed terms. To that end, we include a local closure premise in the rule BETA-STAR-REFL, as shown next.

$$\frac{\mathsf{lc}\, t}{t \longrightarrow^{*} t}\ \textsc{beta-star-refl} \qquad \frac{t \longrightarrow t' \qquad t' \longrightarrow^{*} t''}{t \longrightarrow^{*} t''}\ \textsc{beta-star-head}$$

In the particular case of reasoning on the output of terminating programs in a call-by-value setting, a big-step semantics can be used instead of a small-step semantics. The following judgment, written "$t \Downarrow v$", describes the fact that the term t reduces in big-step towards the value v. It is defined using two inductive rules. The first one states that a value reduces to itself, where the definition of a value is the same as the one used earlier on (see §5.1). The second rule describes the reduction of an application.

$$\frac{\mathsf{value}\ v}{v \Downarrow v}\ \textsc{big-step-val} \qquad \frac{t_1 \Downarrow \mathsf{abs}\, t_3 \qquad t_2 \Downarrow v_2 \qquad {t_3}^{v_2} \Downarrow v_3}{\mathsf{app}\, t_1\, t_2 \Downarrow v_3}\ \textsc{big-step-app}$$

5.3 Simply-typed λ-calculus

In order to provide a complete definition of the typing judgment for the simply-typed λ-calculus, we first need to give a formal definition of simple types and of typing environments. Simple types are built from atomic type and arrow types.

$$T \quad := \quad A \quad | \quad T_1 \to T_2$$

Typing environments, also called typing contexts, associate types with atoms. Environments are constructed from the empty environment, written "$\varnothing$", and by extending an existing environment with a given binding, written "$E,\ x\ :\ T$". Technically, environments are implemented using association lists, of type $\mathsf{list}\,(\mathsf{atom} * T)$. So, $\varnothing$ is represented as nil and "$E,\ x\ :\ T$" is represented as "$(x, T) :: E$". The domain of an environment E, written "$\mathrm{dom}(E)$", corresponds to the set of names that are bound by that environment. It is computed as the set of keys of the corresponding association list.

We require environments to bind names at most once. While this restriction is not strictly necessary for the simply-typed λ-calculus, it is needed for reasoning on more involved systems. For the sake of uniformity, the library for defining environments that we have developed requires environments to bind any given name at most once. The following "ok" predicate captures this property, by enforcing that bindings do not reuse names that are already in the domain of the environment they are appended to.

$$\frac{}{\mathrm{ok}\,\varnothing}\ \textsc{ok-nil} \qquad\qquad \frac{\mathrm{ok}\,E \qquad x \notin \mathrm{dom}(E)}{\mathrm{ok}\,(E,\ x\ :\ T)}\ \textsc{ok-cons}$$

The formal rules defining the typing judgment for the simply-typed λ-calculus in locally nameless style can be stated as follows.

$$\frac{\mathrm{ok}\,E \qquad (x\ :\ T) \in E}{E\ \vdash\ \mathsf{fvar}\,x\ :\ T}\ \textsc{typing-var} \qquad \frac{E\ \vdash\ t_1\ :\ T_1 \to T_2 \qquad E\ \vdash\ t_2\ :\ T_1}{E\ \vdash\ \mathsf{app}\,t_1\,t_2\ :\ T_2}\ \textsc{typing-app}$$

$$\frac{\forall\, x \notin L, \quad E,\ x\ :\ T_1\ \vdash\ t^x\ :\ T_2}{E\ \vdash\ \mathsf{abs}\,t\ :\ T_1 \to T_2}\ \textsc{typing-abs}$$

The regularity lemma associated with this judgment states that whenever a typing relation "$E\ \vdash\ t\ :\ T$" holds, E is a well-formed environment and t is a locally closed term. The proof of this lemma is straightforward by induction.

$$\textsc{typing\_regular}: \qquad E \vdash t : T \quad \Rightarrow \quad \mathrm{ok}\,E \quad \wedge \quad \mathsf{lc}\,t$$

5.4 System F$_{<:}$

This example focuses on System F$_{<:}$. This system is particularly interesting with respect to binding issues, as it mixes two kinds of variables: type variables and term variables.[3] The conventional presentation of the grammars of types, terms and environments is as follows. Types are made of type variables, the maximum type "Top", arrow

[3] The formalization of System F$_{<:}$ and a proof of its soundness are the heart of the POPLMark challenge [Aydemir et al., 2005], which was designed as a good stress test for comparing binding technologies.

types and universal types with bounded quantification. Terms are made of term variables, term abstractions, term applications, type abstractions and type applications. Environments are made of the empty environment, environments extended with term variable bindings and environments extended with type variable bindings.

$$\begin{array}{lcl} T & := & X \;\;|\;\; \mathsf{Top} \;\;|\;\; T \to T \;\;|\;\; \forall X{<:}T.\,T \\ t & := & x \;\;|\;\; \lambda x{:}T.\,t \;\;|\;\; t\,t \;\;|\;\; \Lambda X{<:}T.\,T \;\;|\;\; t\,[T] \\ E & := & \varnothing \;\;|\;\; E,\,x{:}T \;\;|\;\; E,\,X{<:}T \end{array}$$

In order to describe the corresponding grammar in locally nameless syntax, we need to introduce distinct constructors for bound variables and for free variables. Thereafter, four constructors for variables are involved: one for bound type variables ($\mathsf{typ\_bvar}$), one for free type variables ($\mathsf{typ\_fvar}$), one for bound term variables ($\mathsf{trm\_bvar}$) and one for free term variables ($\mathsf{trm\_fvar}$). It is not needed that the atoms used to represent free type variables be different from the atoms used to represent free term variables, as free term variable names can never end up being mixed with free type variable names. Note that universal types, abstractions and type abstractions become nameless.

$$\begin{array}{lcl} T & := & \mathsf{typ\_bvar}\; i \;\;|\;\; \mathsf{typ\_fvar}\; x \;\;|\;\; \mathsf{Top} \;\;|\;\; T \to T \;\;|\;\; \forall{<:}T.\,T \\ t & := & \mathsf{trm\_bvar}\; i \;\;|\;\; \mathsf{trm\_fvar}\; X \;\;|\;\; \lambda{:}T.\,t \;\;|\;\; t\,t \;\;|\;\; \Lambda{<:}T.\,T \;\;|\;\; t\,[T] \end{array}$$

For the sake of presentation of typing and subtyping rules, we introduce the following convention. Whenever we write a lowercase name, it stands for the free term variable with the corresponding name (e.g. "x" stands for "$\mathsf{trm\_fvar}\; x$"), and whenever we write an uppercase name, it stands for the free type variable with the corresponding name (e.g. "X" stands for "$\mathsf{typ\_fvar}\; X$"). Bound variables never appear in typing rules, so there is no need to introduce any particular notation for them.

Two local closure predicates are defined. The first one characterizes locally closed types. It is written "$\mathsf{typ\_lc}\; T$". The second one characterizes locally closed terms. It is written "$\mathsf{trm\_lc}\; t$". Three variable opening operations are required. The first one is used to open universal types, and replaces bound type variables with free type variables inside types. The second one is used to open type abstractions, and replaces bound type variables with free type variables inside terms. The third one is used to open term abstractions, and replaces bound term variables with free term variables insider terms. Similarly, three substitutions are involved: one for substituting types in types, one for substituting types in terms and one for substituting terms in terms. Also, three functions for gathering free variables are defined: one to gather free type variables in types, one to gather free type variables in terms and one for gathering free term variables in terms. Note that variable closing is not needed for establishing the soundness of System $\mathrm{F}_{<:}$.

To formalize the definition of environments, a natural approach would be to introduce an inductive type with three constructors: one for empty environments, one for extensions a with term variable binding and one for extensions with a type variables binding. However, this would prevent us from reusing a standard association lists library, thereby requiring us to duplicate many definitions. Thus, we use a slightly different approach that allows us to define System $\mathrm{F}_{<:}$ environments in terms of association lists. First, we define a binding item, written B, as either a term variable binding or a type variable binding. Both are built upon a type.

$$B \quad := \quad (:T) \;\;|\;\; (<:T)$$

TYPING-VAR
$$\frac{\mathrm{ok}\,E \qquad (x{:}T) \in E}{E \vdash x : T}$$

TYPING-SUB
$$\frac{E \vdash t : S \qquad E \vdash S <: T}{E \vdash t : T}$$

TYPING-ABS
$$\frac{\forall x \notin L, \quad E,\, x{:}T_1 \vdash t^x : T_2}{E \vdash (\lambda{:}T_1.\, t) : (T_1 \to T_2)}$$

TYPING-APP
$$\frac{E \vdash t_1 : T_1 \to T_2 \qquad E \vdash t_2 : T_1}{E \vdash (t_1\, t_2) : T_2}$$

TYPING-TABS
$$\frac{\forall X \notin L, \quad E,\, X{<:}T_1 \vdash t^X : T_2}{E \vdash (\Lambda{<:}T_1.\, t) : (\forall{<:}T_1.\, T_2)}$$

TYPING-TAPP
$$\frac{E \vdash t_1 : (\forall{<:}T_1.\, T_2) \qquad E \vdash T <: T_1}{E \vdash (t_1\, [T_2]) : ({T_2}^{T})}$$

Fig. 1 Typing rules for System F$_{<:}$

The environments that we use are lists of pairs that associate binding items with atoms. An atom bound in a given environment is the name of a free term variable if it is associated with an item of the form $(: T)$, and is the name of a free type variable if it is associated with an item of the form $(<: T)$. As variables are bound at most once in a given environment, a free type variable and a free term variable can never be confused. For the sake of presentation, we let "$E,\, x{:}T$" be a notation for "$(x, (: T)) :: E$" and "$E,\, X{<:}T$" be a notation for "$(X, (<: T)) :: E$".

We need to restrict environments to well-formed ones. An environment is well-formed if all the types that it contains are *well-defined* at their position in the environment. A type T is well-defined in a context E, written "$\mathsf{typ\_wf}\; E\; T$", if T is locally closed and has its free variables bound in the environment E. In the corresponding formal definition, shown next, $\mathsf{typ\_lc}$ is the local closure predicate for types, and $\mathsf{typ\_typ\_fv}$ is the function that computes the set of free types variables occurring in a given type.[4]

$$\mathsf{typ\_wf}\; E\; T \quad \equiv \quad \mathsf{typ\_lc}\; T \quad \wedge \quad \mathsf{typ\_typ\_fv}\; T \subseteq \mathrm{dom}(E)$$

The predicate "ok" captures well-formed typing environments from System F$_{<:}$.

$$\frac{}{\mathrm{ok}\,\varnothing}\ \textsc{ok-nil} \qquad\qquad \frac{\mathrm{ok}\,E \qquad x \notin \mathrm{dom}(E) \qquad \mathsf{typ\_wf}\; E\; T}{\mathrm{ok}\,(E,\, x{:}T)}\ \textsc{ok-cons-typ}$$

$$\frac{\mathrm{ok}\,E \qquad x \notin \mathrm{dom}(E) \qquad \mathsf{typ\_wf}\; E\; T}{\mathrm{ok}\,(E,\, X{<:}T)}\ \textsc{ok-cons-sub}$$

The formal presentation of typing rules for System F$_{<:}$ in locally nameless style are shown in Figure 1. The subtyping rules appear in Figure 2. The three variable opening operations are used in these rules. In the rule TYPING-ABS, a term t is opened with respect to a term variable x. In the rule TYPING-TABS, a term t is opened with respect to a type variable X. In the rule SUB-ALL, a type T is opened with respect to a type variable X. The opening operation is also used for typing type applications in the rule TYPING-TAPP, for reducing a universal type onto a particular argument. Note that only a few well-formedness premises are needed.

The regularity lemmas associated with the typing and the subtyping judgments are stated below. The first one states that if t admits the type T in the environment E,

[4] The proposition "$\mathsf{typ\_wf}\; E\; T$" can also be defined inductively, following the structure of T.

$$\textsc{sub-refl-var}\ \frac{\mathsf{ok}\, E \qquad \mathsf{typ\_wf}\, E\, X}{E \vdash X <: X} \qquad \textsc{sub-trans-var}\ \frac{(X{<:}S) \in E \qquad E \vdash S <: T}{E \vdash X <: T} \qquad \textsc{sub-arrow}\ \frac{E \vdash T_1 <: S_1 \qquad E \vdash S_2 <: T_2}{E \vdash S_1 \to S_2 <: T_1 \to T_2}$$

$$\textsc{sub-top}\ \frac{\mathsf{ok}\, E \qquad \mathsf{typ\_wf}\, E\, T}{E \vdash T <: \mathsf{Top}} \qquad \textsc{sub-all}\ \frac{E \vdash T_1 <: S_1 \qquad (\forall X \notin L,\quad E, X{<:}T_1 \vdash {S_2}^{X} <: {T_2}^{X})}{E \vdash (\forall{<:}S_1.\, S_2) <: (\forall{<:}T_1.\, T_2)}$$

Fig. 2 Subtyping rules for System $\mathrm{F}_{<:}$

then E must be well-formed, T must be well-defined in E, and t must be locally closed. The second one states that if S is a subtype of T in the environment E, then E must be well-formed, and S and T must be well-defined in E.

$$\begin{array}{llclclcl}
\textsc{typing-regular:} & E \vdash t : T & \Rightarrow & \mathsf{ok}\, E & \wedge & \mathsf{trm\_lc}\, t & \wedge & \mathsf{typ\_wf}\, E\, T \\
\textsc{subtyping-regular:} & E \vdash S <: T & \Rightarrow & \mathsf{ok}\, E & \wedge & \mathsf{typ\_wf}\, E\, S & \wedge & \mathsf{typ\_wf}\, E\, T
\end{array}$$

Remark: System F involves two syntactic categories (terms and types), leading to the need for three substitution functions. More generally, the number of substitution functions required grows quadratically with the number of syntactic categories. A technique called *collapsed syntax* [Aydemir et al., 2009] can be employed for reducing the number of substitution functions involved. It consists in collapsing the various syntactic categories into one single category. For example, System F entities can be represented using a single data type that includes both term constructors and type constructors. Note that the number of local closure predicates and of regularity lemmas is not reduced through the use of collapsed syntax.

5.5 Definition of the CPS transformation

Our last example is concerned with the formal definition of a CPS transformation. While previous examples involved only inductive definitions, this example involves the definition of a recursive function on locally nameless terms. One central difficulty here is to find out where variable opening and variable closing operations need to be computed. The textbook definition of the CPS transformation [Plotkin, 1975], written $[\![\cdot]\!]$, is:

$$\begin{array}{lcl}
[\![x]\!] & \equiv & \lambda k.\, k\, x \\
[\![\lambda x.\, t]\!] & \equiv & \lambda k.\, k\, (\lambda x.\, [\![t]\!]) \\
[\![t_1\, t_2]\!] & \equiv & \lambda k.\, [\![t_1]\!]\, (\lambda x.\, [\![t_2]\!]\, (\lambda y.\, x\, y\, k))
\end{array}$$

Translating this definition in locally nameless style takes three steps: first, opening abstractions with fresh names before recursive calls are made on their body; second, closing with respect to the corresponding names the result of those recursive calls; and third, replacing the abstractions that are built for describing continuations with their locally nameless equivalent. The result is shown in Figure 3.[5] The interesting case is the abstraction case: in order to transform an abstraction "$\mathsf{abs}\, t_1$", we pick a name x fresh from t_1, we make a recursive call on the description of its body ${t_1}^{x}$, and then we close the result with respect to variable x and build the result "$\mathsf{abs}\, ({}^{\backslash x}[\![{t_1}^{x}]\!])$".

[5] To implement the recursive function cps in a logic of total functions, one needs to argue that the size of the argument of the function is decreasing at each recursive call. Moreover, when the argument is a bound variable, the function needs to return an arbitrary term.

```
let rec cps $t$ =
  match $t$ with
  | fvar $x$ ↦
    abs (app (bvar 0) (fvar $x$))
  | abs $t_1$ ↦
    let $x$ = fresh (fv($t_1$)) in
    let $t_1'$ = $\backslash^x$(cps ($t_1{}^x$)) in
    abs (app (bvar 0) (abs $t_1'$))
  | app $t_1$ $t_2$ ↦
    let $K$ = abs (app (app (bvar 1) (bvar 0)) (bvar 2)) in
    abs (app (cps $t_1$) (abs (app (cps $t_2$) $K$)))
  | bvar $i$ ↦ undefined
```

Fig. 3 A first implementation of the CPS transformation

```
let rec cps $t$ =
  match $t$ with
  | fvar $x$ ↦
    let $k$ = fresh (fv($t$)) in
    Abs $k$ (app (fvar $k$) (fvar $x$))
  | abs $t_1$ ↦
    let $k, x$ = fresh$_2$ (fv($t$)) in
    let $t_1'$ = $\backslash^x$(cps ($t_1{}^x$)) in
    Abs $k$ (app (fvar $k$) (abs $t_1'$))
  | app $t_1$ $t_2$ ↦
    let $k, k_1, k_2$ = fresh$_3$ (fv($t$)) in
    let $K$ = Abs $k_2$ (app (app (fvar $k_1$) (fvar $k_2$)) (fvar $k$)) in
    Abs $k$ (app (cps $t_1$) (Abs $k_1$ (app (cps $t_2$) $K$)))
  | bvar $i$ ↦ undefined
```

Fig. 4 A second implementation of the CPS transformation, without de Bruijn indices

The definition from Figure 3 is correct and usable. Yet, one could argue that the definition is only partly satisfactory because it involves explicit de Bruijn indices for describing the bound variables introduced by the CPS transformation. Fortunately, there exists an alternative way of writing an equivalent function using only names. The idea is to use a variable closing operation applied to a fresh name. For example, to describe the term "$\lambda k.\,(k\,x)$", we can write "abs ($\backslash^k$(app (fvar k) (fvar x)))" instead of "abs (app (bvar 0) (fvar x))". We can now rewrite the CPS transformation in a more readable style, that closely resemble that of FreshML [Shinwell et al., 2003]. The resulting CPS function, shown in Figure 4, is presented using two pieces of notation. First, following Gordon [1993], we rely on an intermediate notation for building abstraction and write "Abs $x\,t$" instead of "abs ($\backslash^x t$)". Second, we use the notation fresh$_n$ to pick a tuple of n fresh names at once. Although the function from Figure 4 is slightly longer than that of Figure 3, it looks much closer to the textbook presentation.

Remark: the locally nameless implementations of the CPS transformation presented above are not algorithmically efficient: they run in quadratic time while the CPS transformation can be implemented in linear time. It is possible to improve the runtime complexity of the CPS function by delaying variable opening and anticipating variable closing, using contexts that are passed as extra arguments in recursive calls. Yet, for the sole purpose of reasoning on formal definitions, runtime efficiency is not an issue.

6 Formal reasoning in locally nameless style

In this section, we describe how to carry out formal proofs on judgments defined in locally nameless style, focusing on the parts of the reasoning that are specific to that representation. First, we consider a proof of soundness for the call-by-value simply-typed λ-calculus ($\lambda_{\rightarrow}$). We describe proofs that weakening, substitution and reduction preserve typing, as well as a proof of the progress property. Second, we prove transitivity of subtyping in System $\mathrm{F}_{<:}$. This proof illustrates how to combine information coming from two derivations. Then, we focus on a proof of confluence for β-reduction. Finally, we explain how to prove that results of the CPS transformation do not depend upon the names chosen to open abstractions.

6.1 Weakening lemma for $\lambda_{\rightarrow}$

The weakening lemma states that if a term is well-typed in a given environment E then it admits the same type in an extension (E, F) of that context. This result, whose statement appears below, is used in the proof of the substitution lemma. Observe that the target environment (E, F) is required to be well-formed.

$$\text{TYPING\_WEAKEN:} \qquad E \vdash t : T \quad \wedge \quad \mathrm{ok}\,(E, F) \quad \Rightarrow \quad E, F \vdash t : T$$

The statement of this lemma needs to be strengthened before an induction can be carried out, by extending the contexts involved with an extra component.[6] The new statement asserts that typing is preserved when arbitrary bindings are inserted in the middle of the initial environment, provided those bindings do not reuse names that are already bound.

$$\text{TYPING\_WEAKEN':} \qquad E, G \vdash t : T \quad \wedge \quad \mathrm{ok}\,(E, F, G) \quad \Rightarrow \quad E, F, G \vdash t : T$$

The proof goes by induction on the typing derivation. When t is a variable or an application, the proof goes exactly as in a standard textbook presentation. In the case where t is a free variable "$\mathsf{fvar}\, x$", we need to show that if x is bound to T in (E, G) then x is also bound to T to (E, F, G), and this is true. In the case where t is an application "$\mathsf{app}\, t_1\, t_2$", we know that $E, G \vdash t_1 : T_1 \rightarrow T_2$ and that $E, G \vdash t_2 : T_1$. By induction hypotheses applied to these two facts, we derive that the types of t_1 and t_2 are preserved when extending the environment to (E, F, G). We can then apply the typing rule for application to build a proof that the application "$\mathsf{app}\, t_1\, t_2$" admits the type T_2 in the extended environment (E, F, G).

The case where t is an abstraction "$\mathsf{abs}\, t_1$" is more interesting, as it slightly differs from the proof carried out in the named representation. We know that (E, F, G) is well-formed and our induction hypothesis states that there exists a finite set of names L such that, for any x not in L, the proposition "$\mathrm{ok}\,(E, F, G, x : T_1)$" implies "$E, F, G, x : T_1 \vdash t^x : T_2$". The goal to be proved is "$E, F, G \vdash \mathsf{abs}\, t_1 : T$".

Thus, starting from a first instance of the rule TYPING-ABS

$$\frac{\forall x \notin L, \quad E,\, G,\, x : T_1 \vdash t^x : T_2}{E, G \vdash \mathsf{abs}\, t : T_1 \rightarrow T_2}\ \text{TYPING-ABS}$$

[6] It is in fact technically possible to perform an induction directly on the initial statement, but it would require to first prove an auxiliary lemma stating that bindings in the typing context can be permuted, involving overall more work than the approach followed here.

and assuming that (E, F, G) does not contain duplicated bindings, we aim at building another instance of the rule TYPING-ABS of the form:

$$\frac{\forall\, x \notin L', \quad E,\, F,\, G,\, x\, :\, T_1 \,\vdash\, t^x \,:\, T_2}{E, F, G \,\vdash\, \mathsf{abs}\, t \,:\, T_1 \to T_2}\ \text{TYPING-ABS}$$

To that end, we need to find a finite set of names L' such that, for any x not in L', the proposition "$E, F, G, x : T_1 \,\vdash\, t^x \,:\, T_2$" holds. we instantiate L' as the union of L and the domain of (E, F, G). On the one hand, by including in L' the names occurring in L, we acquire direct knowledge about the typing of t^x, from the induction hypothesis. More precisely, we are able to show that the proposition "$E, F, G, x : T_1 \,\vdash\, t^x \,:\, T_2$" holds.[7] On the other hand, by including in L' the set of names bound in (E, F, G), we are able to build the environment "$E,\, F,\, G,\, x \,:\, T_1$" without breaking the invariant that names should be bound at most once in the context. More precisely, we are able to prove "$\mathrm{ok}\,(E, F, G, x : T_1)$", using the fact that (E, F, G) is well-formed and that x is fresh for the domain of (E, F, G).

Technically, it would be sufficient to instantiate L' as the union of L and of the domain of F, since the freshness of x from the domains of E and G can be deduced from the fact that the proposition "$E,\, G,\, x \,:\, T_1 \,\vdash\, t^x \,:\, T_2$" holds. In other words, the set L can be proved to already include the domains of both E and G. Yet, as we are not restricted in the number of names that we include in L', we are not at all interested in minimizing the size of the set L'. In practice, we do exactly the opposite: we include in L' as many names as we can, so as to get the strongest possible freshness hypothesis on the name x.

6.2 Substitution lemma for $\lambda_\rightarrow$

The substitution lemma states that the typing of a term t is preserved when substituting a free variable named z of type U with a term u of the same type U. This lemma plays a central role in the proof that β-reduction preserves typing.

$$\text{TYPING\_SUBST:} \qquad E, z : U \,\vdash\, t \,:\, T \quad \wedge \quad E \,\vdash\, u \,:\, U \quad \Rightarrow \quad E \,\vdash\, [z \to u]\, t \,:\, T$$

The skeleton of proof is quite similar to that of the weakening lemma. The main novelty is the need to permute a variable opening operation with a substitution in the proof case for abstractions. The statement first needs to be generalized with a context extension F before being proved by induction on the typing derivation of the term t.

$$\text{TYPING\_SUBST':} \qquad E, z : U, F \,\vdash\, t \,:\, T \quad \wedge \quad E \,\vdash\, u \,:\, U \quad \Rightarrow \quad E, F \,\vdash\, [z \to u]\, t \,:\, T$$

The variable case and the application case are standard. In the particular case where t is exactly the free variable named z, i.e. the one being substituted, we know that z is bound to T in the context, and the goal is to show "$E, F \,\vdash\, u \,:\, T$". Because names are bound at most once in the context, we deduce that T must be equal to U. The remaining proof obligation "$E, F \,\vdash\, u \,:\, U$" is deduced from the hypothesis "$E \,\vdash\, u \,:\, U$" by application of the weakening lemma. The side-condition from the weakening lemma,

[7] We have implicitly used the fact that environments are associative. Indeed, the conclusion of the induction hypothesis mentions a context of the form "$((E, F), (G, x : T))$" while the goal mentions a context of the form "$(((E, F), G), x : T)$". In a formal proof, this equality needs to be justified through an explicit rewriting step.

"ok (E, F)", can be deduced from the fact that "ok $(E, z : U, F)$". The latter is a consequence of the regularity of the typing judgment "$E, z : U, F \vdash t : T$".

The interesting case with respect to the locally nameless representation occurs when t is an abstraction "$\mathsf{abs}\, t_1$". The induction hypothesis states that there exists a set L such that, for any x not in L, the proposition "$E, F, x : T_1 \vdash [z \to u]\,(t^x) : T_2$" holds. The goal is "$E, F \vdash [z \to u]\,(\mathsf{abs}\, t_1) : T$", which, by definition of substitution (see §3.5), is equivalent to "$E, F \vdash \mathsf{abs}\,([z \to u]\, t_1) : T$". To prove it, we apply the typing rule for abstraction, and need to find a set L' such that, for any x not in L', the proposition "$E, F, x : T_1 \vdash ([z \to u]\, t)^x : T_2$" holds. This latter proposition corresponds to the induction hypothesis if we can derive the following equality:

$$[z \to u]\,(t^x) \;=\; ([z \to u]\, t)^x$$

This equality is exactly the matter of the lemma SUBST_OPEN_VAR (see §3.5). To conclude, we need to verify the two side-conditions associated with that lemma. First, u must be locally closed. This fact can be derived from the regularity of the typing hypothesis on u. Second, x must be distinct from z. To be able to show x and z distinct, it suffices to instantiate L' as $L \cup \{z\}$. Indeed, if x is not in L', and if L' includes z, then x is provably distinct from z.

Including the name z in the set L' is a way to ensure that the name x, which is used to describe the variable bound by the abstraction "$\mathsf{abs}\, t$", is distinct from the free variable z. This corresponds to the intuition that, when working with the named representation, a bound variable can always be assumed to be fresh from any known free variable.

6.3 Preservation lemma for $\lambda_\to$

The preservation lemma states that if a term t admits the type T and reduces to a term t', then t' also admits the same type T.

$$\text{PRESERVATION:} \qquad E \vdash t : T \quad \wedge \quad t \longrightarrow_{\mathsf{cbv}} t' \quad \Rightarrow \quad E \vdash t' : T$$

This lemma is proved by induction on the typing derivation of t, followed in each case with a case analysis on the reduction hypothesis. If t is a variable or an abstraction, then it cannot take a reduction step. If t is an application "$\mathsf{app}\, t_1\, t_2$", three sub-cases are possible, depending on the reduction rule being applied. If the reduction occurs inside t_1 or inside t_2, we conclude using the induction hypotheses.

Otherwise, the reduction is the contraction of a β-redex. In this last case, t must be an application of the form "$\mathsf{app}\,(\mathsf{abs}\, t_3)\, v_2$" and t' is equal to $({t_3}^{v_2})$. The typing hypothesis ensures the existence of a type T' such that "$E \vdash \mathsf{abs}\, t_3 : T' \to T$" and "$E \vdash v_2 : T'$". By inversion on the typing hypothesis for "$\mathsf{abs}\, t_3$", there must exists a set L such that, for any x not in the set L, the proposition "$E, x : T' \vdash {t_3}^x : T$" holds. To summarize, the hypotheses available are the premises of the typing derivation for t:

$$\dfrac{\dfrac{\forall x \notin L,\quad E, x : T' \vdash {t_3}^x : T}{E \vdash \mathsf{abs}\, t_3 : T' \to T}\;\text{TYPING-ABS} \qquad E \vdash v_2 : T'}{E \vdash \mathsf{app}\,(\mathsf{abs}\, t_3)\, v_2 : T}\;\text{TYPING-APP}$$

The goal is to prove that $({t_3}^{v_2})$ admits the type T. In order to invoke the substitution lemma and conclude, we first need to introduce a substitution. So, we pick an arbitrary name x fresh from t and L, and exploit the lemma SUBST_INTRO to change ${t_3}^{v_2}$ into $[x \to v_2]\,({t_3}^x)$. (Note that we here use the fact that x is fresh from t.) We then conclude by invoking the substitution lemma, using the typing assumption for v_2 as well as the typing hypothesis for ${t_3}^x$. (The latter is available because we have picked x outside of L.) To summarize, we have built a typing derivation for t' as follows.

$$\dfrac{\dfrac{E, x : T' \vdash {t_3}^x : T \qquad E \vdash v_2 : T'}{E \vdash [x \to v_2]\,({t_3}^x) : T}\ \textsc{typing-subst}}{E \vdash {t_3}^{v_2} : T}\ \textsc{subst-intro}$$

The key idea from this proof is to reason on the result of a β-reduction (${t_3}^{v_2}$) by introducing a substitution on an arbitrary fresh name x, that is, going through the form $[x \to v_2]\,({t_3}^x)$. By comparison, when working with the named representation, there is no need to introduce a substitution explicitly, because β-reduction is already defined in terms of a substitution.

6.4 Progress lemma for $\lambda_\to$

The progress lemma states that if a term t is well-typed in the empty environment, then either t is a value or t can take a reduction step towards some term t'.

$$\textsc{progress:} \qquad \emptyset \vdash t : T \quad \Rightarrow \quad \mathsf{value}\, t \quad \vee \quad \exists\, t',\ t \longrightarrow_{\mathsf{cbv}} t'$$

There is nothing in this proof specific to the locally nameless representation, except the witness to be provided in the case where a β-reduction can be performed.

The proof goes by induction on the typing derivation. First, as the typing environment is empty, t cannot be a variable. Second, if t is an abstraction, then t is a value. Otherwise, t is an application "$\mathsf{app}\, t_1\, t_2$". If t_1 is not a value, then, by induction hypothesis, it can be reduced. Otherwise, it t_2 is not a value, then, by induction hypothesis, it can be reduced. Otherwise, both t_1 and t_2 are values. Since t_1 has an arrow type, it must be an abstraction, of the form "$\mathsf{abs}\, t_3$". In this case, the β-reduction rule applies, and the application "$\mathsf{app}\,(\mathsf{abs}\, t_3)\, t_2$" reduces to (${t_3}^{t_2}$).

6.5 Transitivity of subtyping in System F$_{<:}$

The proof of soundness of System F$_{<:}$ involves a key intermediate lemma, whose purpose is to establish the transitivity of the subtyping relation.

$$E \vdash S <: T \quad \wedge \quad E \vdash T <: U \quad \Rightarrow \quad E \vdash S <: U$$

The case where S, T and U are universal types is particularly interesting with respect to the treatment of variable bindings, because we need to relate variables coming from two different derivations.

The proof is conducted by induction on the structure of T, that is, following the induction principle associated with the local closure of T. We focus on the case where S, T and U are universal types. The corresponding proof obligation appears next. The first pair of hypotheses come from the subtyping relation stating that S is smaller than T. The second pair come from the fact that T is smaller than U. The goal is to show that S is smaller than U. Notice that each of the two cofinitely-quantified hypotheses come with its own set of excluded names, called L and L'.

$$\begin{array}{ll} & \begin{cases} E \vdash T_1 <: S_1 \\ \forall X \notin L, \quad E, X{<:}T_1 \vdash ({S_2}^X) <: ({T_2}^X) \end{cases} \\ \wedge & \begin{cases} E \vdash U_1 <: T_1 \\ \forall X \notin L', \quad E, X{<:}U_1 \vdash ({T_2}^X) <: ({U_2}^X) \end{cases} \\ \Rightarrow \quad \exists L''. & \begin{cases} E \vdash U_1 <: S_1 \\ \forall X \notin L'', \quad E, X{<:}U_1 \vdash ({S_2}^X) <: ({U_2}^X) \end{cases} \end{array}$$

The first part of the conclusion, which asserts that U_1 smaller than S_1, is an immediate consequence of the induction hypothesis, since U_1 is smaller than T_1 and T_1 is smaller than S_1. For the second part of the conclusion, we instantiate L'' as the union of L and L'. Now, let X be an arbitrary atom not in the set L''. On the one hand, since X is not in L, we know that $({S_2}^X)$ is smaller than $({T_2}^X)$ in the context "$E, X{<:}T_1$". By invoking a *narrowing lemma* (not detailed here), we can show that the same subtyping relation actually holds in the context "$E, X{<:}U_1$". On the other hand, since X is not in L', $({T_2}^X)$ is smaller than $({U_2}^X)$ in the context "$E, X{<:}U_1$". By induction hypothesis applied to those two results, we conclude that $({S_2}^X)$ is smaller than $({U_2}^X)$ in the context "$E, X{<:}U_1$". This completes the proof.

In the above reasoning, we have used one name X to open three binders coming from two different judgments. As each of the two judgments is quantified over its own cofinite set, we need to pick X in the intersection of these two cofinite sets. The reason why the cofinite quantification works here is because the intersection of two cofinite sets always produces a cofinite set. In the proof, we have effectively built this intersection by constructing L'', the finite set of names to be excluded, as the union of L and L'.

6.6 Proof of an introduction lemma

We now describe the proof of an introduction lemma. We consider the cofinitely-quantified inductive rule BETA-ABS, which explains how to reduce an abstraction.

$$\frac{\forall x \notin L, \quad t^x \longrightarrow {t'}^x}{\mathsf{abs}\, t \longrightarrow \mathsf{abs}\, t'} \ \textsc{beta-abs}$$

The corresponding introduction lemma, named BETA-ABS-INTRO, will be useful for our next example (§6.7). It is stated as follows. Note that the freshness conditions on x are necessary: the proposition would not hold if t or t' could contain an occurrence of x.

$$\textsc{beta-abs-intro:} \qquad t^x \longrightarrow {t'}^x \quad \wedge \quad x \# t \quad \wedge \quad x \# t' \quad \Rightarrow \quad \mathsf{abs}\, t \longrightarrow \mathsf{abs}\, t'$$

Let us prove BETA-ABS-INTRO. The goal is to show that $\mathsf{abs}\, t$ reduces to $\mathsf{abs}\, t'$. We apply the cofinitely-quantified inductive rule BETA-ABS. We need to find a set L such that, for any y not in L, the reduction $t^y \longrightarrow {t'}^y$ holds. We instantiate L as the empty set and consider an arbitrary name y. Using the lemma SUBST-INTRO, we introduce a renaming operation from x to y. More precisely, we rewrite t^y as $[x \to y]\,(t^x)$ and symmetrically we rewrite ${t'}^y$ as $[x \to y]\,({t'}^x)$. It remains to establish the following implication:

$$t^x \longrightarrow {t'}^x \qquad \Rightarrow \qquad [x \to y]\,(t^x) \longrightarrow [x \to y]\,({t'}^x)$$

This implication is an immediate consequence of the renaming lemma associated with the β-reduction judgment:

$$\textsc{beta-rename:}\quad u \longrightarrow v \quad\Rightarrow\quad [x \to y]\, u \longrightarrow [x \to y]\, v$$

This result can be easily established by induction on the hypothesis $v \longrightarrow v'$. Only the case where v and v' are abstractions requires some care. The induction hypothesis asserts that $[x \to y]\,(u^z) \longrightarrow [x \to y]\,(v^z)$ holds for any name z not in some set L. We need to establish that $([x \to y]\,u)^z \longrightarrow ([x \to y]\,v)^z$ holds for all names z not in some finite set L'. We instantiate L' as the $L \cup \{x\}$ and conclude using the lemma SUBST_OPEN_VAR to permute the substitutions with the variable opening operations.

The renaming lemma BETA-RENAME is in fact a particular case of a substitution lemma for the β-reduction relation. This substitution lemma, called β-SUBST-OUT (its statement appears in §6.8) is needed anyway in order to establish properties of β-reduction. In many situations like here, we can save the need to perform an induction for proving a renaming lemma by reusing a substitution lemma directly. Note that the proof of a substitution lemma for a given judgment can generally be conducted without help of the renaming lemma associated with that judgment.

6.7 Interaction of binders with the β^*-reduction

The following lemma states that if a body t^x reduces in several steps towards another body t'^x, then the abstraction "$\mathsf{abs}\, t$" reduces to "$\mathsf{abs}\, t'$" in several steps.

$$\beta^*\text{-}\textsc{abs:}\qquad (\forall\, x \notin L,\quad t^x \longrightarrow^* t'^x) \quad\Rightarrow\quad \mathsf{abs}\, t \longrightarrow^* \mathsf{abs}\, t'$$

This lemma is involved in particular to establish confluence of β-reduction (see §6.8). We have chosen to state the lemma using a cofinite quantification in order to obtain a statement that looks similar to the rule BETA-ABS and that does not need to include explicit freshness side-conditions.

That said, in order to prove the lemma, we need to go through an existentially-quantified version of the lemma. Indeed, in order to perform an induction on the reduction sequence starting on t^x, we must settle on one particular name x before performing the induction. Thus, we need to prove the following intermediate lemma.

$$\beta^*\text{-}\textsc{abs-intro:}\qquad t^x \longrightarrow^* t'^x \quad\wedge\quad x \# t, t' \quad\Rightarrow\quad \mathsf{abs}\, t \longrightarrow^* \mathsf{abs}\, t'$$

To prove this lemma, we first reformulate it as follows:

$$u \longrightarrow^* u' \quad\Rightarrow\quad \forall\, t, t',\quad u = t^x \quad\wedge\quad u' = t'^x \quad\wedge\quad x \# t, t' \quad\Rightarrow\quad \mathsf{abs}\, t \longrightarrow^* \mathsf{abs}\, t'$$

We can then perform the induction on the reduction sequence $u \longrightarrow^* u'$. There are two cases. In the first case, suppose that the reduction from u is the empty sequence. In this case, u' is equal to u. In order to conclude, we must show that t is equal to t'. This is an immediate consequence of the injectivity of variable opening (see §3.2). In the second case, suppose there exists a term u'' such that $u \longrightarrow u''$ and $u'' \longrightarrow^* u'$. In other to apply the induction hypothesis on the latter fact, we need to find a term t'' such that $u'' = t''^x$. To that end, we define t'' as $\setminus^x u''$. It remains to show that "$\mathsf{abs}\, t$" reduces to "$\mathsf{abs}\, t'$". We know that t^x reduces to t'^x, but only for one particular name x known to be fresh from t and t'. Thus, we cannot invoke the cofinitely-quantified inductive rule BETA-ABS. Instead, we conclude with the corresponding introduction lemma BETA-ABS-INTRO.

This proof illustrates the fact that a cofinitely-quantified rule is sometimes too weak as an introduction form. In such cases, the associated introduction lemma needs to be explicitly invoked. The above proof also shows a situation where reasoning on the injectivity of variable opening is required. Both these issues do not appear on conventional paper-and-pencil proofs, where abstractions are implicitly α-renamed during inductions, so as to be able to assume sufficient freshness. These issues do not appear either in a proof carried out in pure de Bruijn style, where no name is ever involved in the reasoning. The lemma β^*-ABS is one of the few examples of a proof which is significantly simpler with the pure de Bruijn representation than with the locally nameless representation.

6.8 Confluence of β-reduction

Confluence of β-reduction is a fundamental result from the theory of pure λ-calculus.

$$\beta\_\text{CONFLUENCE:} \quad t \longrightarrow^* t_1 \quad \wedge \quad t \longrightarrow^* t_2 \quad \Rightarrow \quad \exists\, t', \quad t_1 \longrightarrow^* t' \quad \wedge \quad t_2 \longrightarrow^* t'$$

The purpose of this section is to describe the parts of the proofs that are specific to the locally nameless representation of λ-terms.

There are several approaches to establishing confluence. We describe a direct syntactic proof based on parallel reductions. This proof is divided in two parts. The first part consists in establishing that the reflexive-transitive closure of β-reduction is equal to the transitive closure of parallel reduction. The second part consists in proving that parallel reduction satisfies the diamond property, which is a strong form of confluence.

$$\text{PARA\_DIAMOND:} \quad t \twoheadrightarrow t_1 \quad \wedge \quad t \twoheadrightarrow t_2 \quad \Rightarrow \quad \exists\, t', \quad t_1 \twoheadrightarrow t' \quad \wedge \quad t_2 \twoheadrightarrow t'$$

The only part which is really specific to the locally nameless representation lies in the proof of an intermediate lemma used to establish the equivalence between β^*-reduction and parallel reduction. This lemma states that if ${t_1}^x$ reduces to ${t_2}^x$ and u_1 reduces to u_2, then the opening ${t_1}^{u_1}$ reduces to the opening ${t_2}^{u_2}$.

$$\beta^*\_\text{THROUGH:} \quad {t_1}^x \longrightarrow^* {t_2}^x \quad \wedge \quad u_1 \longrightarrow^* u_2 \quad \Rightarrow \quad {t_1}^{u_1} \longrightarrow^* {t_2}^{u_2} \quad \text{when } x \# t_1, t_2$$

To prove this result, we introduce two substitutions to decompose the two opening operations, in a similar fashion as done in the proof of PRESERVATION (see §6.3). More precisely, we introduce a fresh name x, and decompose ${t_1}^{u_1}$ as $[x \to u_1]\,({t_1}^x)$, and, symmetrically, decompose ${t_2}^{u_2}$ as $[x \to u_2]\,({t_2}^x)$. The remaining result to be proved is a form of substitution lemma for the relation β^*.

$$\beta^*\_\text{SUBST\_ALL:} \quad t \longrightarrow^* t' \quad \wedge \quad u \longrightarrow^* u' \quad \Rightarrow \quad [x \to u]\, t \longrightarrow^* [x \to u']\, t'$$

The proof of this lemma goes by induction on the reduction sequence starting on t, and involves two auxiliary lemmas, which are stated and proved next.

The first auxiliary lemma states that β-reduction is preserved through substitution of a free variable with an arbitrary locally closed term.

$$\beta\_\text{SUBST\_OUT:} \quad t \longrightarrow t' \quad \wedge \quad \mathsf{lc}\, u \quad \Rightarrow \quad [x \to u]\, t \longrightarrow [x \to u]\, t'$$

Its proof goes by induction on the first hypothesis. All the cases are easy, except the case where a β-redex is contracted. In this case, we need to show that $[x \to u]\,(\mathsf{app}\,(\mathsf{abs}\,t_1)\,t_2)$ reduces to $[x \to u]\,({t_1}^{t_2})$. By definition of substitution, the former term is equal to "$\mathsf{app}\,(\mathsf{abs}\,([x \to u]\,t_1))\,([x \to u]\,t_2)$", and thus reduces towards $([x \to u]\,t_1)^{([x\to u]\,t_2)}$. It remains to argue that this latter expression is equal to $[x \to u]\,({t_1}^{t_2})$. This distributivity property of substitution on opening is exactly the matter of the lemma SUBST_OPEN, described in §3.7.

The second auxiliary lemma describes how β-reduction steps can be "plugged into" a given λ-term through a substitution. More precisely, if u reduces to u', then, given a locally closed term t, the substitution of x with u in t produces a term that reduces to the substitution of x with u' in t.

$$\beta^*\text{\_SUBST\_IN:}\quad u \longrightarrow^* u' \quad\wedge\quad \mathsf{lc}\,t \quad\Rightarrow\quad [x \to u]\,t \;\longrightarrow^*\; [x \to u']\,t$$

The proof of this lemma goes by induction on the structure of t, i.e., by induction on the proof of the local closure of t. If t is a variable, the result is immediate. Otherwise, we need to show that the β^*-relation commutes with the application and the abstraction constructors:

$$\begin{array}{llllll}
\beta^*\text{-APP-1:} & t_1 \longrightarrow^* t_1' & \wedge & \mathsf{lc}\,t_2 & \Rightarrow & \mathsf{app}\,t_1\,t_2 \longrightarrow^* \mathsf{app}\,t_1'\,t_2 \\
\beta^*\text{-APP-2:} & t_2 \longrightarrow^* t_2' & \wedge & \mathsf{lc}\,t_1 & \Rightarrow & \mathsf{app}\,t_1\,t_2 \longrightarrow^* \mathsf{app}\,t_1\,t_2' \\
\beta^*\text{-ABS:} & (\forall\, x \notin L,\;\; t^x \longrightarrow^* t'^x) & & & \Rightarrow & \mathsf{abs}\,t \longrightarrow^* \mathsf{abs}\,t'
\end{array}$$

The two first results are easy. The last one has been established earlier on, in §6.7.

A comparison between a confluence proof in de Bruijn style and the same proof in locally nameless style shows that the two proofs have about the same size and have relatively similar structures. The two main causes of divergence are the treatment of abstraction cases on the one hand, and the fact that the de Bruijn presentation involves shifting operations in many statements and proofs on the other hand. The first cause leads a few auxiliary lemmas, such as β^*-ABS, to require a slightly longer proof in the locally nameless development. The second cause leads the pure de Bruijn development to be further apart from a conventional presentation than the locally nameless development.

6.9 Properties of the CPS transformation

The implementation of the CPS transformation on locally nameless terms presented in §5.5 can be formally proved to preserve the semantics of the terms it transforms. The purpose of this section is not to present the complete proof, but only to describe how to establish that results of CPS transformations do not depend on the arbitrary names being used to open abstractions. The need for reasoning on the irrelevance of local variable names comes from the fact that CPS is defined as a function, which precludes the use of a cofinite quantification. The key intermediate lemma is:

$$\text{CPS\_OPEN\_VAR:}\qquad {}^{\backslash x}(\mathsf{cps}(t^x)) \;=\; {}^{\backslash y}(\mathsf{cps}(t^y)) \qquad \text{when } x \mathrel{\#} t \wedge y \mathrel{\#} t \wedge \mathsf{body}\,t$$

The left-hand side of the above equality describes a call to cps for the term t^x, while the right-hand side describes a call on the term t^y. The two terms only differ by a renaming of a free variable: the second term is equal to a copy of the first one in which all the occurrences of the name x have been replaced with the name y. Thus, in order to

$$\textsc{cps\_rename}: \quad \mathsf{cps}([x \to y]\, t) \;=\; [x \to y]\, (\mathsf{cps}\ t) \qquad \text{when } y \# t \wedge \mathsf{lc}\, t$$

The proof goes by induction on the size of t. Only the case where t is an abstraction is nontrivial. In this case, we must show that $[x \to y]\, (\backslash^{a}(\mathsf{cps}(t^{a})))$ is equal to $\backslash^{b}(\mathsf{cps}(([x \to y]\, t)^{b}))$, where a is an atom fresh for t and b is an atom fresh for $[x \to y]\, t$. Because neither a and b are fresh from both t and $[x \to y]\, t$, we need to pick a third atom c, fresh from x, y, a, b and t. This freshness allows us to justify the following series of rewriting steps.

$$\begin{array}{lll}
 & [x \to y]\, (\backslash^{a}(\mathsf{cps}(t^{a}))) & & \\
= & [x \to y]\, (\backslash^{c}([a \to c]\, (\mathsf{cps}(t^{a})))) & \text{by } \textsc{close-val-rename}, & \text{since } c \# \mathsf{cps}(t^{a}) \\
= & [x \to y]\, (\backslash^{c}(\mathsf{cps}(([a \to c]\, t^{a})))) & \text{by induction hypothesis,} & \text{since } c \# t^{a} \\
= & [x \to y]\, (\backslash^{c}(\mathsf{cps}(t^{c}))) & \text{by } \textsc{subst-intro}, & \text{since } a \# t \\
= & \backslash^{c}([x \to y]\, (\mathsf{cps}(t^{c}))) & \text{by } \textsc{subst-close-var}, & \text{since } c \# x, y \\
= & \backslash^{c}(\mathsf{cps}([x \to y]\, (t^{c}))) & \text{by induction hypothesis,} & \text{since } y \# t^{c} \\
= & \backslash^{c}(\mathsf{cps}(([x \to y]\, t)^{c})) & \text{by } \textsc{subst-open-var}, & \text{since } c \# x \\
= & \backslash^{b}([c \to b]\, (\mathsf{cps}(([x \to y]\, t)^{c}))) & \text{by } \textsc{close-var-rename}, & \text{since } b \# \mathsf{cps}(([x \to y]\, t)) \\
= & \backslash^{b}(\mathsf{cps}([c \to b]\, (([x \to y]\, t)^{c}))) & \text{by induction hypothesis,} & \text{since } b \# ([x \to y]\, t)^{c} \\
= & \backslash^{b}(\mathsf{cps}(([x \to y]\, t)^{b})) & \text{by } \textsc{subst-intro}, & \text{since } c \# [x \to y]\, t
\end{array}$$

Fig. 5 Proof of a renaming lemma for a function on locally nameless terms

establish the above equality, we need to exploit the fact that the cps function commutes with renaming (lemma CPS_RENAME, stated further on). More precisely, the proof of CPS_OPEN_VAR goes as follows:

$$\begin{array}{lll}
 & \backslash^{y}(\mathsf{cps}(t^{y})) & \\
= & \backslash^{y}(\mathsf{cps}([x \to y]\, (t^{x}))) & \text{by } \textsc{subst-intro} \\
= & \backslash^{y}([x \to y]\, (\mathsf{cps}(t^{x}))) & \text{by } \textsc{cps\_rename} \\
= & \backslash^{x}(\mathsf{cps}(t^{x})) & \text{by } \textsc{close\_var\_rename}
\end{array}$$

It remains to explain how to prove that cps commutes with renaming.

$$\textsc{cps\_rename}: \quad \mathsf{cps}([x \to y]\, t) \;=\; [x \to y]\, (\mathsf{cps}\ t) \qquad \text{when } y \# t \wedge \mathsf{lc}\, t$$

We prove by induction on the size of the term t that, for any names x and y, the cps function distributes over the renaming of x into y. The details of the proof appears in Figure 5. Two basic properties of the cps function are also needed in the verification of the CPS transformation: it preserves local closure and it preserves the set of free variables. These two facts can be easily proved by induction.

$$\begin{array}{lllll}
\textsc{cps\_lc}: & \mathsf{lc}\, t & & \Rightarrow & \mathsf{lc}\, (\mathsf{cps}\ t) \\
\textsc{cps\_fv}: & \mathsf{lc}\, t & \wedge \quad x \# t & \Rightarrow & x \# (\mathsf{cps}\ t)
\end{array}$$

7 Advanced binding structures

All the binders considered so far in the paper are *simple binders*: they just bind one name at a time in a given body. This section discusses the representation of multiple binders, pattern matching structures and recursive binders. The manipulation of these advanced forms of binding structures involves lists of fresh atoms, so we start by introducing notation for describing such lists.

7.1 Lists of fresh atoms

Let overlined symbols denote lists of values. For example, $\overline{x}$ stands for a list of atoms and $\overline{t}$ stands for a list of terms. The notation $|\overline{x}|$ denotes the length of the list $\overline{x}$. The constant nil denotes the empty list, and $x :: \overline{y}$ denotes the consing of x to the list $\overline{y}$.

We introduce a proposition, written "$\mathsf{distinct}\ L\ n\ \overline{x}$", to capture the property that $\overline{x}$ is a list of n pairwise-distinct atoms that are all fresh from the set L. The predicate $\mathsf{distinct}$ can be implemented in several ways. We have found it convenient to use the following inductive definition.

$$\frac{}{\mathsf{distinct}\ L\ 0\ \mathsf{nil}}\ \textsc{distinct-nil} \qquad\qquad \frac{x \# L \qquad \mathsf{distinct}\ (L \cup \{x\})\ n\ \overline{y}}{\mathsf{distinct}\ L\ (n+1)\ (x :: \overline{y})}\ \textsc{distinct-cons}$$

Assuming we have a fresh name generator, we can build an *iterated fresh name generator*, called $\mathsf{fresh\_list}$. Given a list L and a natural number n, the function $\mathsf{fresh\_list}$ returns a list $\overline{x}$ made of n distinct atoms that are all fresh for L. More formally, if $\overline{x}$ is defined as "$\mathsf{fresh\_list}\, L\, n$", then the proposition "$\mathsf{distinct}\ L\ n\ \overline{x}$" is satisfied.

In the case of simple binders, the cofinite quantification takes the form "$\forall\, x,\ x \notin L \Rightarrow P\, x$", where P is some predicate. Following mathematical presentation, we have abbreviated this statement as "$\forall\, x \notin L,\ P\, x$". In the case of multiple binders, the cofinite quantification takes the form "$\forall\, \overline{x},\ \mathsf{distinct}\ L\ n\ \overline{x} \Rightarrow P\, \overline{x}$". Similarly, we introduce an abbreviation: we write "$\forall\, \overline{x}^n \notin L,\ P\, x$" as a shorthand for that statement.

7.2 Multiple bindings

To present the way multiple binders can be handled in locally nameless style, we extend the grammar of λ-terms with a let construct that binds several names at once. In ML, this binding form is typically written "$\mathsf{let}\, x_1 = t_1\ \mathsf{and}\ \ldots\ \mathsf{and}\, x_n = t_n\ \mathsf{in}\, t'$". For the sake of presentation, we abbreviate the construction as "$\mathsf{let}\, \overline{x} = \overline{t}\ \mathsf{in}\, t'$". This simple construction suffices to illustrate the treatment of multi-binders.

When working with multi-binders, there are two possibilities for representing bound variables. One possibility is to treat a multi-binder binding n variables exactly as a sequence of n abstractions. Yet, this approach is not very practical to work with. In particular, when the opening or the substitution function reaches a multi-binder, it needs to augment the current depth by the number of variables that are bound by that multi-binder. To avoid such arithmetic operations, we chose to follow another approach. We represent bound variables with two natural numbers: the first number is a de Bruijn index describing to which multi-binder the variable is bound, while the second number is an index used to distinguish between the variables bound by a same multi-binder. (Such use of pairs of indices to represent multi-binders is well-known to experts of pure de Bruijn syntax.)

The grammar of λ-terms extended with the multiple let binder appears next. Observe that bound variables are represented with two indices and that free variables are still represented using a single atom.

$$t \quad := \quad \mathsf{bvar}\, i\, j \quad | \quad \mathsf{fvar}\, x \quad | \quad \mathsf{abs}\, t \quad | \quad \mathsf{app}\, t\, t \quad | \quad \mathsf{let}\, \overline{t}\, t$$

Simple binders, such as abstraction, are viewed as multiple binders that bind only one variable. The bound variable "$\mathsf{bvar}\, i\, j$" refers to the (i+1)-th enclosing binder, which can be either an abstraction or a let. If it refers to a let, then the value j indicates which of the variables bound by the let is being referred to. If it refers to an abstraction, then j must be equal to zero (this invariant will be enforced by the local closure predicate).

The opening operation, written $t^{\overline{u}}$, replaces all the variables bound to a multi-binder with terms taken from a list. The term t is the body of the multi-binder being opened and $\overline{u}$ is the list of values that are to be substituted for the variables bound by the multi-binder. The variable opening operation replaces a bound variable "$\mathsf{bvar}\, i\, j$" with the j-th value from the list $\overline{u}$ when the de Bruijn index i matches the current depth. Multiple-opening is defined in terms of an auxiliary recursive function, written $\{k \to \overline{u}\}\, t$.

$$\begin{array}{lcl} \multicolumn{1}{r}{t^{\overline{u}}} & \equiv & \{0 \to \overline{u}\}\, t \\ \{k \to \overline{u}\}\, (\mathsf{bvar}\, i\, j) & \equiv & \text{if } (i = k) \text{ then } (\mathsf{List.nth}\, j\, \overline{u}) \text{ else } (\mathsf{bvar}\, i\, j) \\ \{k \to \overline{u}\}\, (\mathsf{fvar}\, y) & \equiv & \mathsf{fvar}\, y \\ \{k \to \overline{u}\}\, (\mathsf{app}\, t_1\, t_2) & \equiv & \mathsf{app}\, (\{k \to \overline{u}\}\, t_1)\, (\{k \to \overline{u}\}\, t_2) \\ \{k \to \overline{u}\}\, (\mathsf{abs}\, t) & \equiv & \mathsf{abs}\, (\{(k+1) \to \overline{u}\}\, t) \\ \{k \to \overline{u}\}\, (\mathsf{let}\, \overline{t}\, t_1) & \equiv & \mathsf{let}\, (\mathsf{List.map}\, (\{k \to \overline{u}\}\, \cdot)\, \overline{t})\, (\{(k+1) \to \overline{u}\}\, t_1) \end{array}$$

Note that the call to $\mathsf{List.nth}$ in the case of bound variables is always applied to a valid index when working on locally closed terms (the definition of which appears next). The recursive calls on the arguments of let are made through a $\mathsf{List.map}$ operation applied to the function that maps a term t to the term $\{k \to \overline{u}\}\, t$.

The local closure predicates ensures that all bound variables are actually bound. The definition in the case of multi-binders generalizes that of simple binders. The body of an abstraction is opened with a list made of a single fresh name. The body of a let-binder is opened with a list of fresh names whose length is equal to the number of arguments of that let.

$$\frac{}{\mathsf{lc}\, (\mathsf{fvar}\, x)}\ \text{LC-VAR} \qquad \frac{\mathsf{lc}\, t_1 \qquad \mathsf{lc}\, t_2}{\mathsf{lc}\, (t_1\, t_2)}\ \text{LC-APP} \qquad \frac{\forall\, x \notin L, \quad \mathsf{lc}\, (t^{x::\mathsf{nil}})}{\mathsf{lc}\, (\mathsf{abs}\, t)}\ \text{LC-ABS}$$

$$\frac{\mathsf{List.forall}\, (\mathsf{lc}\, \cdot)\, \overline{t} \qquad (\forall\, \overline{x}^{|\overline{t}|} \notin L, \quad \mathsf{lc}\, (t_1{}^{\overline{x}}))}{\mathsf{lc}\, (\mathsf{let}\, \overline{t}\, t_1)}\ \text{LC-LET}$$

The definition of the predicate "body" also needs to be generalized. The proposition "$\mathsf{bodies}\, n\, t$" asserts that t becomes a locally closed term when opened with n names.

$$\mathsf{bodies}\, n\, t \quad \equiv \quad \exists\, L, \ \forall\, \overline{x}^{n} \notin L, \quad \mathsf{lc}\, (t^{\overline{x}})$$

The semantics of multi-binders can be defined using the multiple opening operation. The rules describing the contraction of abstraction and of let-bindings appear next.

$$\frac{\mathsf{bodies}\, 1\, t \qquad \mathsf{lc}\, u}{\mathsf{app}\, (\mathsf{abs}\, t)\, u \longrightarrow t^{u::\mathsf{nil}}}\ \text{BETA-RED-ABS} \qquad \frac{\mathsf{List.forall}\, (\mathsf{lc}\, \cdot)\, \overline{t} \qquad \mathsf{bodies}\, |\overline{t}|\, t_1}{\mathsf{let}\, \overline{t}\, t_1 \longrightarrow t_1{}^{\overline{t}}}\ \text{BETA-RED-LET}$$

A typing rules for let-bindings is shown below. It involves adding several bindings at once to the typing context. If $\overline{x}$ is a list of distinct fresh names and $\overline{T}$ is a list of types of the same length, we write $(E, \overline{x : T})$ the extension of the environment E with all the bindings from the list obtained by pairing items from $\overline{x}$ with items from $\overline{T}$.

$$\frac{\begin{array}{c} \forall\, (t, T) \in (\mathsf{List.combine}\, \overline{t}\, \overline{T}), \quad E \vdash t : T \\ \forall\, \overline{x}^{|\overline{t}|} \notin L, \quad E, \overline{x : T} \vdash t_1{}^{\overline{x}} : T_1 \end{array}}{E \vdash \mathsf{let}\, \overline{t}\, t_1 : T_1}\ \text{TYPING-LET}$$

7.3 Pattern matching

The manipulation of pattern matching constructions also relies on a multiple opening function. In what follows, we explain how the locally nameless representation can handle linear patterns (where each bound variable may occur at most once) and non-linear patterns (where a same variable may occur several times). As running example, we consider the syntax of λ-terms extended with binary pairs, binary sums and pattern-destructuring abstractions. The key idea is to represent variables of a pattern with de Bruijn indices when that pattern is used to bind variables, and to describe the same pattern using names when reasoning on the pattern itself. In a sense, we apply the locally nameless representation to patterns.

The grammar of terms and patterns appears below. The constructor for abstraction is built upon a pattern and a term. The constructor for pairs is written pair and the constructor for injections is written inj^k, where k is equal to either 1 or 2 (indicating whether the term is a left or a right injection). The grammar of patterns includes constructors for bound variables, for free variables, for pairs and for injections. It also includes wildcard, constants, as well as constructors for describing conjunction and disjunction of patterns. Remark: alias-patterns, written "p as x" in Caml and "x as p" in SML, can be viewed as a particular case of conjunction-patterns.

$$\begin{array}{lll}
t & := & \mathsf{bvar}\,i\,j \;\mid\; \mathsf{fvar}\,x \;\mid\; \mathsf{abs}\,p\,t \;\mid\; \mathsf{app}\,t\,t \;\mid\; \mathsf{pair}\,t\,t \;\mid\; \mathsf{inj}^k\,t \\
p & := & \mathsf{pbvar}\,j \;\mid\; \mathsf{pfvar}\,x \;\mid\; \mathsf{ppair}\,p\,p \;\mid\; \mathsf{pinj}^k\,p \;\mid \\
 & & \mathsf{pwild} \;\mid\; \mathsf{pconst}\,t \;\mid\; \mathsf{pand}\,p\,p \;\mid\; \mathsf{por}\,p\,p
\end{array}$$

The meta-variable j ranges over the indices used to identify pattern variables. Intuitively, if a pattern p binds n variables, then the indices of the pattern variables should range over the set $[0, n[$. In the particular case of linear patterns, each index from the range $[0, n[$ should appear exactly once in the pattern. Henceforth, the arity n of a pattern p is written $||p||$.

In many programming languages, constants occurring in patterns are not allowed to contain any free variable. However, if we allow pattern expression of the form "$\mathsf{pconst}\,t$" to appear with a non-closed term t, then the functions manipulating syntax (opening, closing, substitution and the free-variable function) need to be extended so as to traverse patterns and work on the terms occurring inside patterns.

The variable opening operation for patterns turns bound pattern variables into free pattern variables. This operation, written $p^{\overline{x}}$, is defined as follows.

$$\begin{array}{lll|lll}
(\mathsf{pbvar}\,j)^{\overline{x}} & \equiv & \mathsf{List.nth}\,j\,\overline{x} & (\mathsf{pwild})^{\overline{x}} & \equiv & \mathsf{pwild} \\
(\mathsf{pfvar}\,y)^{\overline{x}} & \equiv & \mathsf{pfvar}\,y & (\mathsf{pconst}\,t)^{\overline{x}} & \equiv & \mathsf{pconst}\,t \\
(\mathsf{ppair}\,p_1\,p_2)^{\overline{x}} & \equiv & \mathsf{ppair}\,(p_1{}^{\overline{x}})\,(p_2{}^{\overline{x}}) & (\mathsf{pand}\,p_1\,p_2)^{\overline{x}} & \equiv & \mathsf{pand}\,(p_1{}^{\overline{x}})\,(p_2{}^{\overline{x}}) \\
(\mathsf{pinj}^k\,p)^{\overline{x}} & \equiv & \mathsf{pinj}^k\,(p^{\overline{x}}) & (\mathsf{por}\,p_1\,p_2)^{\overline{x}} & \equiv & \mathsf{por}\,(p_1{}^{\overline{x}})\,(p_2{}^{\overline{x}})
\end{array}$$

Note that $\mathsf{List.nth}$ is always applied onto a valid index when the pattern involved in the variable opening operation is well-formed.

A closed pattern p is a *well-formed closed pattern*, written "$\mathsf{pattern}\ p$", if and only if the variable opening of the pattern p with a list $\overline{x}$ of fresh names returns a *well-formed opened pattern* whose free variables are exactly those in the list $\overline{x}$. To describe well-formed opened patterns, we introduce a judgment, written "$\mathsf{binds}\ p\ S$", which states that the opened pattern p binds exactly the names in the set S. In particular, this judgment ensures that no bound variable remains in an opened pattern and that two branches of any disjunction pattern bind the same set of names. We start with the definition of the binding judgment for non-linear patterns, in which a same variable may be bound several times.

$$\overset{\textsc{binds-var}}{\frac{}{\mathsf{binds}\ (\mathsf{pfvar}\,x)\ \{x\}}} \qquad \overset{\textsc{binds-pair}}{\frac{\mathsf{binds}\ p_1\ S_1 \qquad \mathsf{binds}\ p_2\ S_2}{\mathsf{binds}\ (\mathsf{ppair}\,p_1\,p_2)\ (S_1 \cup S_2)}} \qquad \overset{\textsc{binds-inj}}{\frac{\mathsf{binds}\ k\ S}{\mathsf{binds}\ (\mathsf{pinj}^k\ p)\ S}} \qquad \overset{\textsc{binds-wild}}{\frac{}{\mathsf{binds}\ (\mathsf{pwild})\ \emptyset}}$$

$$\overset{\textsc{binds-const}}{\frac{\mathsf{lc}\,t}{\mathsf{binds}\ (\mathsf{pconst}\,t)\ \emptyset}} \qquad \overset{\textsc{binds-and}}{\frac{\mathsf{binds}\ p_1\ S_1 \qquad \mathsf{binds}\ p_2\ S_2}{\mathsf{binds}\ (\mathsf{pand}\,p_1\,p_2)\ (S_1 \cup S_2)}} \qquad \overset{\textsc{binds-or}}{\frac{\mathsf{binds}\ p_1\ S \qquad \mathsf{binds}\ p_2\ S}{\mathsf{binds}\ (\mathsf{por}\,p_1\,p_2)\ S}}$$

For linear patterns, we need to ensure that each pattern variable occurs at most once. To that end, we enforce the unions of set of names to be *disjoint* unions. Only the rules for pairs and for conjunctions need to be extended with a disjointness premise.

$$\frac{\mathsf{binds}\ p_1\ S_1 \qquad \mathsf{binds}\ p_2\ S_2 \qquad S_1 \cap S_2 = \emptyset}{\mathsf{binds}\ (\mathsf{ppair}\,p_1\,p_2)\ (S_1 \cup S_2)}\ \textsc{binds'-pair}$$

$$\frac{\mathsf{binds}\ p_1\ S_1 \qquad \mathsf{binds}\ p_2\ S_2 \qquad S_1 \cap S_2 = \emptyset}{\mathsf{binds}\ (\mathsf{pand}\,p_1\,p_2)\ (S_1 \cup S_2)}\ \textsc{binds'-and}$$

We now define well-formed patterns: p is well-formed, written "$\mathsf{pattern}\ p$", if and only if the opening $p^{\overline{x}}$ binds exactly the variables $\overline{x}$. This definition involves a cofinite quantification of the list of variables $\overline{x}$. Observe that the proposition "$\mathsf{pattern}\ p$" cannot hold if the pattern p contains any sub-pattern of the form "$\mathsf{pfvar}\,y$" for some name y.

$$\mathsf{pattern}\ p \qquad \equiv \qquad \exists\, L, \quad \forall\, \overline{x}^{||p||} \notin L, \quad \mathsf{binds}\ (p^{\overline{x}})\ (\mathsf{List.to\_set}\,\overline{x})$$

The definition of local closure of terms captures well-formedness of patterns occurring inside them. The local closure rule for pattern-destructuring abstraction is:

$$\frac{\mathsf{pattern}\ p \qquad (\forall\, \overline{x}^{||p||} \notin L, \quad \mathsf{lc}\,(t^{\overline{x}}))}{\mathsf{lc}\,(\mathsf{abs}\,p\,t)}\ \textsc{lc-abs}$$

It remains to explain how to state the semantics and typing rules for pattern matching. To describe the semantics, we introduce a relation describing successful pattern matching, written "$\mathsf{match}\ p\ t\ M$", which relates an opened pattern p, a term t, and an instantiation map M mapping atoms to terms. The inductive definition of this judgment appears in Figure 6. The definition involves a judgment "$\mathsf{compatible}\ M_1\ M_2$". For non-linear patterns, "$\mathsf{compatible}\ M_1\ M_2$" should be defined so as to capture that the two maps M_1 and M_2 agree on the intersection of their domain. For linear patterns, "$\mathsf{compatible}\ M_1\ M_2$" should be defined so as to capture that the two maps M_1 and M_2 have disjoint domains.

Note that we have added local closure assumptions in the premises of MATCH-WILD and MATCH-CONST, as well as extra hypotheses in the MATCH-OR rules, in order to ensure regularity. The regularity lemma associated with the pattern matching judgment "$\mathsf{match}\ p\ t\ M$" states that the term t involved is locally closed, that the pattern p binds exactly the variables that are in the domain of M and that the map M binds terms that are locally closed.

$$\textsc{match\_regular}: \quad \mathsf{match}\,p\,t\,M \ \Rightarrow\ \mathsf{lc}\,t \ \wedge\ \mathsf{binds}\,p\,(\mathrm{dom}(M)) \ \wedge\ (\forall\,(x,u) \in M,\ \ \mathsf{lc}\,u)$$

$$\textsc{match-var}\quad \frac{\mathsf{lc}\, t}{\mathsf{match}\ (\mathsf{pfvar}\, x)\ t\ (\{x \rightsquigarrow t\})} \qquad \textsc{match-pair}\quad \frac{\mathsf{match}\ p_1\ t_1\ M_1 \qquad \mathsf{match}\ p_2\ t_2\ M_2 \qquad \mathsf{compatible}\ M_1\ M_2}{\mathsf{match}\ (\mathsf{ppair}\, p_1\, p_2)\ (\mathsf{pair}\, t_1\, t_2)\ (M_1 \cup M_2)}$$

$$\textsc{match-inj}\quad \frac{\mathsf{match}\ p\ t\ M}{\mathsf{match}\ (\mathsf{pinj}^k\ p)\ (\mathsf{inj}^k\ t)\ M} \qquad \textsc{match-wild}\quad \frac{\mathsf{lc}\, t}{\mathsf{match}\ (\mathsf{pwild})\ t\ \emptyset} \qquad \textsc{match-const}\quad \frac{\mathsf{lc}\, t}{\mathsf{match}\ (\mathsf{pconst}\, t)\ t\ \emptyset}$$

$$\textsc{match-or-left}\quad \frac{\mathsf{match}\ p_1\ t\ M \qquad \mathsf{binds}\ p_2\ S_2}{\mathsf{match}\ (\mathsf{por}\, p_1\, p_2)\ t\ M} \qquad \textsc{match-or-right}\quad \frac{\mathsf{match}\ p_2\ t\ M \qquad \mathsf{binds}\ p_1\ S_1}{\mathsf{match}\ (\mathsf{por}\, p_1\, p_2)\ t\ M}$$

$$\textsc{match-and}\quad \frac{\mathsf{match}\ p_1\ t\ M_1 \qquad \mathsf{match}\ p_2\ t\ M_2 \qquad \mathsf{compatible}\ M_1\ M_2}{\mathsf{match}\ (\mathsf{pand}\, p_1\, p_2)\ t\ (M_1 \cup M_2)}$$

Fig. 6 Judgment describing successful pattern matching

$$\textsc{pat-var}\quad \frac{\mathrm{ok}\, E \qquad (x : T) \in E}{E \vdash (\mathsf{pfvar}\, x) : T} \qquad \textsc{pat-wild}\quad \frac{\mathrm{ok}\, E}{E \vdash (\mathsf{pwild}) : T} \qquad \textsc{pat-const}\quad \frac{E \vdash t : T}{E \vdash (\mathsf{pconst}\, t) : T} \qquad \textsc{pat-inj}\quad \frac{E \vdash p : T_k}{E \vdash (\mathsf{pinj}^k\ p) : (T_1 + T_2)}$$

$$\textsc{pat-pair}\quad \frac{E \vdash p_1 : T_1 \qquad E \vdash p_2 : T_2}{E \vdash (\mathsf{ppair}\, p_1\, p_2) : (T_1 \times T_2)} \qquad \textsc{pat-and}\quad \frac{E \vdash p_1 : T \qquad E \vdash p_2 : T}{E \vdash (\mathsf{pand}\, p_1\, p_2) : T} \qquad \textsc{pat-or}\quad \frac{E \vdash p_1 : T \qquad E \vdash p_2 : T}{E \vdash (\mathsf{por}\, p_1\, p_2) : T}$$

Fig. 7 Typing judgment for pattern matching

Using the pattern matching judgment, we can state the reduction rules for patterns. Consider the application of an abstraction "$\mathsf{abs}\, p\, t$" onto an argument u. This application reduces towards $t^{\overline{v}}$, where $\overline{v}$ is the list of subterms of u that are bound to variables from the pattern p. In the corresponding reduction rule shown below, $\overline{x}$ is a list of fresh names used to open the pattern p.

$$\frac{\mathsf{lc}\,(\mathsf{abs}\, p\, t) \qquad (\forall\, \overline{x}^{||p||} \notin L,\ \ \mathsf{match}\ p\ u\ \{\overline{x} \rightsquigarrow \overline{v}\})}{\mathsf{app}\,(\mathsf{abs}\, p\, t)\, u \longrightarrow t^{\overline{v}}}\ \textsc{beta-pattern}$$

Remark: another way to guarantee the local closure premise "$\mathsf{lc}\,(\mathsf{abs}\, p\, t)$" is to require both "$\mathsf{pattern}\ p$" and "$\mathsf{bodies}\ ||p||\ t$".

Finally, we present the typing judgment for patterns and the typing rule for pattern matching abstractions. The typing judgment for patterns takes the form "$E \vdash p : T$", where p is an opened pattern, T is a type and E is a typing environment. Its inductive definition appears in Figure 7.

The typing rule for pattern matching abstraction is defined as follows: if the pattern p opened with fresh names $\overline{x}$ has the type T_1 and the term t opened with fresh names $\overline{x}$ has the type T_2 under the assumption that the free variables named $\overline{x}$ have the type $\overline{T}$, then "$\mathsf{abs}\, p\, t$" admits the type "$T_1 \to T_2$".

$$\frac{\mathsf{pattern}\ p \qquad (\forall\, \overline{x}^{||p||} \notin L, \quad \overline{x : T} \vdash p^{\overline{x}} : T_1 \quad \wedge \quad E, \overline{x : T} \vdash t^{\overline{x}} : T_2)}{E \vdash \mathsf{abs}\, p\, t : T_1 \to T_2}\ \textsc{typing-abs}$$

Recursive bindings occur for instance in the representation of recursive functions or recursive types. We start with simple recursive binders, and then explain how to support mutually-recursive structures.

7.4.1 Simple recursive types

Consider a constructor for simple recursive type, written "$\mu X.T$" with the named representation. The meaning is that the name X is bound in the body T to the recursive type "$\mu X.T$" itself. The unfolding operation on such a recursive type consists in replacing occurrences of the variable X with copies of the type "$\mu X.T$" inside the body T.

Such recursive types can be modelled in the locally nameless representation using a constructor "$\mathsf{rec}\,T$", which is a simple binder that binds one variable in its body T. The unfolding operation can be modelled by opening the body T with the type itself. In the following rule, the symbol $\sim$ stands for equivalence between types.

$$\frac{}{\mathsf{rec}\,T \quad\sim\quad T^{(\mathsf{rec}\,T)}}\ \textsc{rec-type-unfold}$$

7.4.2 Simple recursive functions

Consider a constructor for recursive function, written "$\mathsf{fix}\,f\,x\ :=\ t$" with the named representation. The meaning is that the name f, which stands for the recursive function itself, and the name x, which stands for the argument, are bound in the body t. The β-reduction rule states that when such a function is applied to a value u, it reduces towards the term "$[x \to u]\,[f \to (\mathsf{fix}\,f\,x\ :=\ t)]\,t$". We can model this kind of recursive function in the locally nameless representation using a constructor "$\mathsf{fix}\,t$", which is a multi-binder that binds two variables in its body t.

The term "$\mathsf{fix}\,t$" is locally closed if and only if its body t is locally closed when opened with a list $\overline{x}$ made of two names.

$$\frac{\forall\,\overline{x}^2 \notin L,\quad \mathsf{lc}\,(t^{\overline{x}})}{\mathsf{lc}\,(\mathsf{fix}\,t)}\ \textsc{lc-fix}$$

The reduction rule states that the application of "$\mathsf{fix}\,t$" to a value u reduces to the opening of the body t with a list made of u and of the fixpoint itself.

$$\frac{\mathsf{bodies}\,2\,t \qquad \mathsf{lc}\,u}{\mathsf{app}\,(\mathsf{fix}\,t)\,u\ \longrightarrow\ t^{(u::\mathsf{fix}\,t::\mathsf{nil})}}\ \textsc{beta-red-fix}$$

The typing rule for fixed points introduces two variables in the typing context, one for the function and one for its argument. There are two ways of presenting this typing rule, depending on whether one quantifies over two names one after the other, or directly over lists of names of length 2. The two approaches are equivalent.

$$\frac{\forall\,f\ \notin\ L,\ \forall\,x\ \notin\ (L \cup \{f\}),\quad E,\,f : (T_1 \to T_2),\,x : T_1\ \vdash\ t^{(x::f::\mathsf{nil})}\ :\ T_2}{E\ \vdash\ \mathsf{fix}\,t\ :\ T_1 \to T_2}\ \textsc{typing-fix}$$

$$\frac{\forall\,\overline{y}^2 \notin L,\quad E,\overline{y : ((T_1 \to T_2) :: T_1 :: \mathsf{nil})}\ \vdash\ t^{\overline{y}}\ :\ T_2}{E\ \vdash\ \mathsf{fix}\,t\ :\ T_1 \to T_2}\ \textsc{typing-fix'}$$

Remark: in the first rule, it is also possible to quantify the variables over two different cofinite sets, i.e. writing "$\forall f \notin L,\ \forall x \notin L',\ \ldots$". However, in practice, it is usually more convenient to instantiate only one set.

7.4.3 Mutually recursive values

The representation of mutually-recursive values is slightly more complex. The representation of a value that has been defined through a mutually-recursive definition needs to carry the definitions of the other values that it depend upon. We can extend the grammar of λ-terms with mutually-defined terms by introducing a constructor "$\mathsf{mut}\,j\,\bar{t}$", where $\bar{t}$ is a list of recursive definitions, and j is an index describing which of these definition corresponds to the current value. Again, we are adapting a standard trick traditionally associated with the pure de Bruijn representation.

The constructor mut behaves as a multi-binder: it binds n variables in each term from the list $\bar{t}$, where n is the length of $\bar{t}$. Thereby, each of the n definitions can refer to any other definition, including itself. A term "$\mathsf{mut}\,j\,\bar{t}$" is locally closed if each term in $\bar{t}$ is locally closed when opened with n names, and if j is a valid index.

$$\dfrac{0 \le j < |\bar{t}| \qquad \forall\, \overline{x}^{|\bar{t}|} \notin L,\quad \forall\, t_i \in \bar{t},\quad \mathsf{lc}\,(t_i^{\overline{x}})}{\mathsf{lc}\,(\mathsf{mut}\,j\,\bar{t})}\ \textsc{lc-mut}$$

In order to unfold a mutually-recursive definition "$\mathsf{mut}\,j\,\bar{t}$", we need to open the j-th body from $\bar{t}$ with a list of terms. The i-th term from that list should correspond to the i-th definition, that is, to "$\mathsf{mut}\,i\,\bar{t}$". So, we introduce an intermediate definition, written $\langle t \rangle$, to describe the list of arguments to be used in the opening operation.

$$\langle\bar{t}\rangle \quad\equiv\quad (\mathsf{mut}\,0\,\bar{t}) :: (\mathsf{mut}\,1\,\bar{t}) :: \ldots :: (\mathsf{mut}\,(n-1)\,\bar{t}) :: \mathsf{nil} \qquad \mathsf{where}\ n = |\bar{t}|$$

The unfolding rule can now be stated. Below, the symbol $\sim$ stands for equivalence between terms.

$$\dfrac{}{\mathsf{mut}\,j\,\bar{t} \quad\sim\quad (\mathsf{List.nth}\,j\,\bar{t})^{\langle\bar{t}\rangle}}\ \textsc{mut-unfold}$$

The β-reduction rule for reducing a mutually-defined function on an argument is a particular case of unfolding:

$$\dfrac{\mathsf{lc}\,(\mathsf{mut}\,j\,\bar{t}) \qquad \mathsf{lc}\,u \qquad (\mathsf{List.nth}\,j\,\bar{t})^{\langle\bar{t}\rangle} = \mathsf{abs}\,v}{\mathsf{app}\,(\mathsf{mut}\,j\,\bar{t})\,u \;\longrightarrow\; v^{u}}\ \textsc{beta-red-mut}$$

To type-check a mutually-recursive definition "$\mathsf{mut}\,j\,\bar{t}$", we introduce n names in the typing context: one for each definition involved in the mutual recursion. These names are bound to n types, described by a list $\overline{T}$. The i-th term from the list $\bar{t}$, written t_i, must admit the i-th type from the list $\overline{T}$, written T_i. The type of "$\mathsf{mut}\,j\,\bar{t}$" is the j-th type from the list $\overline{T}$.

$$\dfrac{\forall\, \overline{x}^{|\bar{t}|} \notin L, \quad \forall\, i, \quad E, \overline{x : T} \vdash t_i^{\overline{x}} : T_i}{E \vdash \mathsf{mut}\,j\,\bar{t} : T_j}\ \textsc{typing-mut}$$

8 A short history of the locally nameless representation

8.1 Names for globally-bound variables, de Bruijn indices for locally-bound variables

The description of a mixed representation using de Bruijn indices for bound variables and names for free variables is as old as the introduction of *nameless dummies*, now most-commonly called *de Bruijn indices*. Indeed, de Bruijn mentions in his founding paper the possibility for such a mixed syntax, while describing an algorithm for turning namefree terms into name-carrying terms [de Bruijn, 1972]. However, de Bruijn does not discuss this mixed representation any further.

The combination of de Bruijn indices with names appears in early implementations of proof assistants: in Paulson's implementation of Isabelle [Paulson, 1986, 1988], as well as in Huet's *Constructive Engine* [Huet, 1989]. The latter served as a starting point for the implementation of the proof assistants Coq [Coq Development Team, 2009] and Lego [Luo and Pollack, 1992]. This representation strategy can also be found in various implementations of HOL, e.g. HOL 4 [Norrish and Slind, 2007], and is briefly described in Paulson's book *ML for the working programmer* [1991].

The main motivations for this mixed representation are simplicity and efficiency. On the one hand, globally-bound variables and constants, which appear in the environment, are represented using names. Therefore, environments can be implemented using hashtables, and thereby support efficient look-up operations. On the other hand, locally-bound variables, which correspond to bindings inside terms, are represented using de Bruijn indices. This saves the need for dealing with α-conversion when comparing terms and allows exploiting sharing of subterms in algorithms. However, in the implementation of proof assistants, binders are not systematically opened when traversed, so the technique used is not, strictly speaking, the locally nameless representation. For example, the algorithm for testing convertibility of two terms is performed in pure de Bruijn's style, traversing abstractions and products without opening their body. The design decision of not opening binders systematically seems to be motivated by the need for efficiency. More recently, Epigram [Altenkirch et al., 2005], an experimental dependently-typed language, has been implemented using the locally nameless representation. In the implementation, terms are manipulated using an adaptation of Huet's Zipper [1997] to locally nameless syntax [McBride and McKinna, 2004].

8.2 Locally nameless terms for reasoning on name-carrying terms

Gordon [1993] appears to be the first to have used the locally nameless representation for the purpose of carrying out formal proofs. He used locally nameless terms (which, somewhat confusingly, he calls *de Bruijn terms*) for the purpose of formally justifying the widely-accepted idea that one can reason on name-carrying terms and at the same time identify terms up to α-conversion. His operations on terms are simplified versions of those found in Paulson's book [1991]: binders are systematically opened when traversed, thus shifting of de Bruijn indices is never necessary.

More precisely, Gordon defines a grammar of locally nameless terms, together with opening (called *instantiate*) and variable closing (called *abstract*). He implements substitution in terms of variable closing and opening (recall the rule subst_as_close_open from §3.7), and defines locally closed terms (called *proper* terms) in terms of a degree function. He then defines name-carrying abstractions in terms of variable closing

("$\mathsf{Abs}\, x\, t$" stands for "$\mathsf{abs}\,(\backslash^{x} t)$") and gives a characterization of λ-terms through a set of rules that closely resemble the rules defining local closure. The main difference is the treatment of abstractions, for which he uses a "forward" inductive rule whose premise is "$\mathsf{lc}\, t$" and whose conclusion is "$\mathsf{lc}\,(\mathsf{Abs}\, x\, t)$".

Gordon then defines the set of conventional name-carrying λ-terms as the set of terms satisfying local closure. Alpha-conversion, which states that the term "$\lambda x.\, t$" is *equal* to the term "$\lambda y.\, ([x \to y]\, t)$" is justified by the lemma CLOSE_VAR_RENAME. A number of lemmas describe how substitution distributes over the constructors. Other lemmas help reasoning by case analysis and by induction on λ-terms. Gordon and Melham [1996] later built upon Gordon's work to justify the soundness of their "Five axioms of alpha-conversion", providing an abstract axiomatic representation of quotiented name-carrying λ-terms.

Gordon's construction involves a lot of infrastructure. Gordon argues that this work needs be done only once and forall, because all binding constructions can be encoded into the pure λ-calculus. Yet, it seems that reasoning through such an encoding is not as easy and lightweight as it sounds. To the extent of our knowledge, no large-scale formalization has been carried out that way.

8.3 Formal reasoning on *locally named* syntax

Pollack built the proof system LEGO using Huet's Constructive Engine as a starting point [Pollack, 1994b]. While working on the theory of LEGO, he wanted to give a formal justification to the core part of that theory. This lead him, together with McKinna, to formalize Pure Type Systems (PTS) within LEGO itself [McKinna and Pollack, 1993]. In order to avoid de Bruijn indices, they used the locally named representation, where both bound variables and free variables are represented with names.

The locally named representation shares a lot of features with the locally nameless representation. The locally named syntax involves a substitution for bound variables and a substitution for free variables. It also includes a judgment similar to local closure: a term is said to be *variable closed* if it contains no unmatched bound variable name. McKinna and Pollack [1993, 1999] studied in details the problem of the quantification of free variable names. While they use existentially-quantified rules for their initial definitions, they show their definitions equivalent to judgments featuring universally-quantified rules (see §4.2). The universal quantification leads to strong induction and inversion principles. For the introduction form, they rely on the inductive rule from the existentially-quantified judgment. This technique provides the desired introduction and elimination form, but requires a very large amount of infrastructure.

Pollack later suggested that the techniques developed for the locally named representation would also work well with what he called the *locally nameless* representation. He described typing rules for a Constructive Engine for PTS in that style, using both the existential and the universal versions of inductive definitions [Pollack, 1994a].

8.4 Formal reasoning in locally nameless style

The POPLMark challenge [Aydemir et al., 2005] has been proposed to stimulate progress on the topic of formalizing definitions of programming languages and checking proofs of their properties. The core of the challenge, a formalization of the soundness of System $F_{<:}$, as been designed to stress many of the critical issues involved for formalizing languages, in particular issues related to the treatment of variable bindings.

Through several talks related to the POPLMark topic, Pollack has emphasized the benefits of the locally nameless representation over other first-order representations of syntax (e.g., [Pollack, 2006]). Leroy attended one of these talks. Soon afterwards, he completed a solution to the first half of the challenge using the Coq proof assistant (October 2005). Later, he addressed the second half of the challenge [Leroy, 2007]. His solution is quite close to the formalization of System $F_{<:}$ presented in this paper, with the exception of the representation of environments and the quantification of free variable names in inductive definitions. Leroy states inductive definitions using universal quantification and then derives existentially-quantified introduction lemmas. Yet, deriving introduction lemmas from universally-quantified definitions is much harder than deriving them from cofinitely-quantified definitions [Aydemir et al., 2008]. In particular, Leroy's submission included dozens of lemmas for showing all definitions and relations stable through permutation of names.

Nevertheless, the locally nameless representation appeared as an appealing approach compared to other techniques based on first-order representations. In the following year, several researchers submitted variations on Leroy's development. Chlipala [2006] re-implemented it with more aggressive proof automation. Charguéraud [2006] redesigned it with cofinite quantification. Ricciotti [2007] ported the proof towards the Matita proof assistant. Together with Aydemir, Pierce, Pollack and Weirich, the author later carried out further investigations on the locally nameless representation, focusing in particular on the cofinite quantification and on the development of practical techniques for working with the locally nameless style [Aydemir et al., 2008].

9 Conclusion

Through this paper, we have described in details the working of the locally nameless representation. We have explained that there are a few cases where a proof in locally nameless style is slightly more involved than it would have been in pure de Bruijn style, because one many need to manually instantiate induction hypotheses or to derive existentially-quantified versions of inductive rules. However, we have found those cases to be relatively rare in practice. Overall, we believe that the locally nameless representation with cofinite quantification is an effective approach to formal metatheory.

The techniques described in this paper have been put to practice through several large-scale developments. In particular, we have formalized type soundness results for System $F_{<:}$ and for ML extended with references, exceptions, datatypes, recursion and pattern-matching. We have also proved the Church-Rosser theorem for pure λ-calculus and proved type preservation for the Calculus of Construction.

Other researchers have also employed the locally nameless representation to formalize results from their research papers, using either Coq or Isabelle/HOL. Many of them were able to build their development on top of one of the four developments mentioned in the previous paragraph. A non-exhaustive list appears next.

- de Vries et al. [2007] proved type soundness for *uniqueness typing*.
- Jia et al. [2008] proved soundness and decidability of type-checking of AURA, a programming language for access control.
- Benton and Koutavas [2007] formalized a bisimulation for the ν-calculus, a simply-typed lambda calculus with fresh name generation.

- Swamy and Hicks [2008] prove type soundness of λAIR, a language that combines dependent, affine and singleton types to enforce information release policies.
- Pratikakis et al. [2008] formalized type soundness of a "contextual effects" system.
- Yakobowski [2008] formalized type soundness for a preliminary version of $x\mathrm{ML}^{\mathrm{F}}$, which is a type system that aims at integrating ML-style type inference in System F.
- Russo and Vytiniotis [2009] formalized QML, a type system where explicit System F types do coexist with ordinary ML types.
- Rendel et al. [2009] formalized F^*_ω, an extension of F_ω that allows typed self-representations (representations of programs inside the programming language).
- Garrigue [2009] formalized a type-checker and an interpreter for the core ML language extended with structural polymorphism and recursion.
- Rossberg et al. [2010] formalized the soundness of an elaboration from ML with modules towards F_ω.
- Henrio et al. [2010] formalized type soundness and Church-Rosser property for the σ-calculus, a theory of objects.
- Papakyriakou et al. [2010] formalized a lambda calculus with impredicative polymorphism and mutable references.
- Effinger-Dean and Grossman [2010] formalized a shared-memory, multi-threaded programming languages with relaxed memory consistency models.
- Montagu [2010] formalized type soundness of Core F-zip, a foundation for a module system, based on a variant of System F where existential types have an open scope.
- Krebbers [2010] formalized $\mathit{\Gamma}_\infty$, a presentation of type theory without explicit contexts, establishing that PTS derivations can be translated into $\mathit{\Gamma}_\infty$ derivations.
- Zhao et al. [2010] formalized System F°, an extension of System F that uses kinds to distinguish linear from intuitionistic terms. They established soundness and completeness of logical equivalence with respect to contextual equivalence.

We hope that the reader will join those researchers and build formal proofs using the locally nameless representation.

Acknowledgments

This work originates in an internship in 2006 on the topic of the POPLMark challenge. I am most grateful to Stephanie Weirich and Benjamin Pierce for their fruitful guidance. I also wish to thank my other co-authors Brian Aydemir and Randy Pollack for their contribution in the writing of the paper *Engineering Formal Metatheory.* Finally, I wish to thank François Pottier and the anonymous reviewers for their useful comments.

References

Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Available from `http://www.e-pig.org/community.html`, 2005.

B. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, GeoffrG.ey Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *TPHOLs*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.

B. Aydemir, S. Weirich, and S. Zdancewic. Abstracting syntax. *Technical Reports (CIS)*, 2009.

Brian Aydemir, Arthur Chargéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2008.

Bruno Barras and Benjamin Werner. Coq in coq. Available from `http://pauillac.inria.fr/~barras/coq_work-eng.html`, 1997.

N. Benton and V. Koutavas. A mechanized bisimulation for the nu-calculus, 2007.

Arthur Charguéraud. Submission to the PoplMark challenge, part 1a. Available from `http://arthur.chargueraud.org/research/2006/poplmark/`, 2006.

Arthur Charguéraud. A collection of formal developments in locally nameless style, March 2009. Scripts from `http://arthur.chargueraud.org/projects/binders/`.

Adam Chlipala. Submission to the PoplMark challenge, part 1a. Available from `http://www.cs.berkeley.edu/~adamc/poplmark/`, 2006.

The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*, 2009. At `http://coq.inria.fr/`.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *IFL*, volume 5083 of *LNCS*, pages 201–218. Springer, 2007.

L. Effinger-Dean and D. Grossman. Modular Metatheory for Memory Consistency Models, 2010.

J. Garrigue. A Certified Interpreter for ML with Structural Polymorphism, 2009.

Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Higher-order Logic Theorem Proving And Its Applications, Proceedings*, volume 780 of *LNCS*, pages 414–426. Springer, 1993.

Andrew D. Gordon and Tom Melham. Five axioms of alpha-conversion. In *TPHOLs*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.

L. Henrio, F. Kammüller, B. Lutz, and H. Sudhof. Locally Nameless Sigma Calculus, 2010.

Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. Functional Pearl.

Gérard Huet. The constructive engine. In *A Perspective in Theoretical Computer Science: Commerative Volume for Gift Siromoney.* World Scientific Publishing, 1989. Also available as INRIA Technical Report 110.

L. Jia, J.A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ACM SIGPLAN International Conference on Functional Programming*, pages 27–38. ACM, 2008.

R. Krebbers. A formalization of Γ_∞ in Coq, 2010. URL `http://robbertkrebbers.nl/research/gammainf/`.

L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

Xavier Leroy. A locally nameless solution to the poplmark challenge. Technical Report 6098, INRIA, January 2007.

Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

Conor McBride and James McKinna. Functional pearl: I am not a number—I am a free variable. In *ACM SIGPLAN Workshop on Haskell*, pages 1–9. ACM Press, 2004.

James McKinna and Robert Pollack. Pure Type Systems formalized. In *Typed Lambda Calculi and Applications: International Conference on Typed Lambda Calculi and Applications, TLCA '93*, volume 664 of *LNCS*, pages 289–305. Springer, 1993.

James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, 1999.

Benoît Montagu. Mechanizing Core F-zip using the locally nameless approach. *5th ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2010.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant For Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Michael Norrish and Konrad Slind. HOL 4. Available from `http://hol.sourceforge.net/`, 2007.

Michalis A. Papakyriakou, Prodromos E. Gerakios, and Nikolaos S. Papaspyrou. A mechanized proof of type safety for the λ-calculus with references, 2010.

Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

Lawrence C Paulson. A preliminary user's manual for isabelle. Technical Report TR-133, Computer Laboratory, University of Cambridge, May 1988.

Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

Randy Pollack. Reasoning about languages with binding: Can we do it yet?, February 2006. Slides from `http://homepages.inf.ed.ac.uk/rpollack/`.

Robert Pollack. Closure under alpha-conversion. In *TYPES'93: Workshop on Types for Proofs and Programs, Nijmegen, May 1993, Selected Papers*, volume 806 of *LNCS*, pages 313–332. Springer, 1994a.

Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994b.

Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks, and Iulian Neamtiu. Formalizing soundness of contextual effects. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 262–277. Springer, 2008.

T. Rendel, K. Ostermann, and C. Hofer. Typed self-representation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 293–303. ACM, 2009.

Wilmer Ricciotti. Submission to the PoplMark challenge, part 1a. Available from `http://ricciott.web.cs.unibo.it/`, 2007.

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Workshop on Types in Language Design and Implementation*, pages 89–102. ACM, 2010.

C.V. Russo and D. Vytiniotis. QML: explicit first-class polymorphism for ML. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 3–14. ACM, 2009.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. FreshML: programming with binders made simple. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 263–274. ACM, 2003.

Nikhil Swamy and Michael Hicks. Verified enforcement of stateful information release policies. *SIGPLAN Notices*, 43(12):21–31, 2008.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automatic Reasoning*, 40:327–356, May 2008.

B. Yakobowski. *Graphical types and constraints: second-order polymorphism and inference*. PhD thesis, Université Paris-Diderot, 2008.

Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In *APLAS*, volume 6461 of *LNCS*, pages 344–359. Springer, 2010.