A verified abstract machine for functional coroutines

Tristan Crolard
CNAM, France
tristan.crolard@cnam.fr

March 4, 2015

Abstract

Functional coroutines are a restricted form of control mechanism, where each continuation comes with its local environment. This restriction was originally obtained by considering a constructive version of Parigot's classical natural deduction which is sound and complete for the Constant Domain logic. In this article, we present a refinement of de Groote's abstract machine which is proved to be correct for functional coroutines. Therefore, this abstract machine also provides a direct computational interpretation of the Constant Domain logic.

Contents

1 Introduction

The *Constant Domain* logic (CD) is a well-known intermediate logic due to Grzegorczyk [25] which can be characterized as a logic for Kripke frames with constant domains. Although CD is semantically simpler than intuitionistic logic, its proof theory is quite difficult : no *conventional* cut-free axiomatization is known [31], and it took more than three decades to prove that the interpolation theorem does not hold either [33]. However, CD is unavoidable when the object of study is *duality in intuitionistic logic*. Indeed, consider the following schema (called either D [25] or DIS [37]), where x does not occur free in B:

$$\forall x(A \vee B) \vdash (\forall x A) \vee B$$

The dual of this schema is $(\exists x A) \wedge B \vdash \exists x(A \wedge B)$ which is clearly valid in intuitionistic logic. Thus bi-intuitionistic logic (also called Heyting-Brouwer logic [37] or subtractive logic [11]), which contains both intuitionistic logic and dual intuitionistic logic, includes both schemas.

Görnemann proved that the addition of the DIS-schema to intuitionistic predicate logic is sufficient to axiomatize CD [23] (and also that the disjunction and existence properties hold, so CD is still a constructive logic). Moreover, Rauszer proved that bi-intuitionistic is conservative over CD [37]. As a consequence, we should expect at least the same difficulties with the proof-theoretical study of bi-intuitionistic logic as with CD. In particular, if we want to understand the computational content of bi-intuitionistic logic, it is certainly worth spending some time on CD.

Although there is no conventional cut-free axiomatization of CD, there are some non-conventional deduction systems which do enjoy cut elimination. The first such system was defined by Kashima [27] as a restriction of Gentzen's sequent calculus LK based on dependency relations. Independently, we described [9] a similar restriction using Parigot's classical natural deduction [35] instead of LK. Another difference lies in the fact that our restriction can also be formulated at the level of proof-terms (terms of the $\lambda\mu$-calculus in Parigot's system), independently of the typing derivation. Such proof-terms, which are terms of Parigot's $\lambda\mu$-calculus, are called *safe* in our calculus [12]. The intuition behind this terminology is presented informally in the introduction of this article as follows:

> "[...] we observe that in the restricted $\lambda\mu$-calculus, even if continuations are no longer first-class objects, the ability of context-switching remains (in fact, this observation is easier to make in the framework of abstract state machines). However, a context is now a pair $\langle$*environment*, *continuation*$\rangle$. Note that such a pair is exactly what we expect as the context of a coroutine, since a coroutine should not access the local environment (the part of the environment which is not shared) of another coroutine. Consequently, we say that a $\lambda\mu$-term t is *safe with respect to coroutine contexts* (or

just *safe* for short) if no coroutines of t access the local environment of another coroutine."

In this paper, we provide some evidence to support this claim in the framework of abstract state machines. As a starting point, we take an environment machine for the $\lambda\mu$-calculus, which is defined and proved correct by de Groote [18] (a very similar machine was defined independently by Streicher and Reus [40]). Then we define a new variant of this machine dedicated to the execution of safe terms (which works exactly as hinted above). Note that this modified machine is surprisingly simpler than what we would expect form the negative proof-theoretic results. We actually obtain a direct, meaningful, computational interpretation of the Constant Domain logic, even though dependency relations were at the beginning only a complex technical device.

As usual with environment machines, it was more convenient to encode variables as de Bruijn indices. Since safe $\lambda\mu$-terms have different scoping rules than regular $\lambda\mu$-terms, the translation into de Bruijn terms should yield different terms: safe $\lambda\mu$-terms need to use *local indices* to access the local environment of the current coroutine, whereas regular terms use the usual *global indices* to access the usual global environment.

As a consequence of this remark, we obtain a proof of correctness of the modified machine which is two-fold. We first introduce an intermediate machine *with local indices, global environment and indirection tables*, then we define two functional bi-simulations $(-)^\star$ and $(-)^\diamond$ showing that this intermediate machine:

- bi-simulates the regular machine with global indices
- bi-simulates the modified machine with local environments

The combined bi-simulation then show that the modified machine is correct with respect to de Groote's regular machine:

$$\begin{array}{lccccccccc}
\text{Regular machine} & \tilde{\sigma}_0^\star & \rightsquigarrow & \cdots & \rightsquigarrow & \tilde{\sigma}_n^\star & \rightsquigarrow & \tilde{\sigma}_{n+1}^\star & \rightsquigarrow & \cdots \\
 & \uparrow \star & & & & \uparrow \star & & \uparrow \star & & \\
\text{Intermediate machine} & \tilde{\sigma}_0 & \rightsquigarrow & \cdots & \rightsquigarrow & \tilde{\sigma}_n & \rightsquigarrow & \tilde{\sigma}_{n+1} & \rightsquigarrow & \cdots \\
 & \downarrow \diamond & & & & \downarrow \diamond & & \downarrow \diamond & & \\
\text{Modified machine} & \tilde{\sigma}_0^\diamond & \rightsquigarrow & \cdots & \rightsquigarrow & \tilde{\sigma}_n^\diamond & \rightsquigarrow & \tilde{\sigma}_{n+1}^\diamond & \rightsquigarrow & \cdots
\end{array}$$

The plan of the paper is the following. In Section 2, we first recall the notion of safety [9], and then we present a simpler (but equivalent) definition of safety which is more convenient for the proofs of correctness. In Section 3, we present the regular and the modified machine. Finally, in Section 4, we detail the proof of correctness: we describe the intermediate machine and the two functional bi-simulations (all the proofs were mechanically checked with the Coq proof assistant).

1.1 Related works

Computational interpretation of classical logic

Since Griffin's pioneering work [24], the extension of the well-known formulas-as-types paradigm to classical logic has been widely investigated for instance by Murthy [34], Barbanera and Berardi [2], Rehof and Sørensen [38], de Groote [19], and Krivine [30]. We shall consider here Parigot's $\lambda\mu$-calculus mainly because it is confluent and strongly normalizing in the second order framework [35]. Note that Parigot's original CND is a second-order logic, in which $\vee, \wedge, \exists, \exists^2$ are definable from $\rightarrow, \forall, \forall^2$. An extension of CND with primitive conjunction and disjunction has also been investigated by Pym, Ritter and Wallen [36] and de Groote [19].

The computational interpretation of classical logic is usually given by a λ-calculus extended with some form of control (such as the famous **call/cc** of Scheme or the catch/throw mechanism of Lisp) or similar formulations of first-class continuation constructs. Continuations are used in denotational semantics to describe control commands such as jumps. They can also be used as a programming technique to simulate backtracking and coroutines. For instance, first-class continuations have been successfully used to implement Simula-like cooperative coroutines in Scheme [22]. This approach has been extended in the Standard ML of New Jersey (with the typed counterpart of Scheme's **call/cc** [26]) to provide simple and elegant implementations of light-weight processes (or threads), where concurrency is obtained by having individual threads voluntarily suspend themselves [39]. The key point in these implementations is that control operators make it possible to switch between coroutine contexts, where the context of a coroutine is encoded as its continuation.

Coroutines

The concept of coroutine is usually attributed to Conway [8] who introduced it to describe the interaction between a lexer and a parser inside a compiler. They are also used by Knuth [29] which sees them as a mechanism that generalizes subroutines (procedures without parameters). Coroutines first appeared in the language Simula-67 [15]. A formal framework for proving the correctness of simple programs containing coroutines has also been developed [7]. Coroutine mechanisms were later introduced in several programming language, for instance in Modula-2 [42], and more recently in the functional language Lua [21].

In his thesis, Marlin [32] summarizes the characteristics of a coroutine as follows:

1. *the values of data local to a coroutine persist between successive occasions on which control enters it (that is, between successive calls), and*
2. *the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.*

That is, a coroutine is a subroutine *with a local state* which can suspend and resume execution. This informal definition is of course not sufficient to capture

the various implementations that have been developed in practice. To be more specific, the main differences between coroutine mechanisms can be summarized in as follows [20]:

- *the control-transfer mechanism, which can provide symmetric or asymmetric coroutines.*
- *whether coroutines are provided in the language as first-class objects, which can be freely manipulated by the programmer, or as constrained constructs;*
- *whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from with nested calls.*

Symmetric coroutines generally offer a single control-transfer operation that allows coroutines to pass control between them. Asymmetric control mechanisms, sometimes called *semi-coroutines* [14], rely on two primitives for the transfer of control: the first to invoke a coroutine, the second to pause and return control to the caller.

A well-known illustration of the third point above, called "the same-fringe problem", is to determine whether two trees have exactly the same sequence of leaves using two coroutines, where each coroutine recursively traverses a tree and passes control to other coroutine when it encounters a leaf. The elegance of this algorithm lies in the fact that each coroutine uses its own stack, which permits for a simple recursive tree traversal. Note however that there are also other solutions of this problem which do not rely on coroutines [5].

Asymmetric coroutines often correspond to the coroutines mechanism made directly accessible to the programmer (as in Simula or Lua), sometimes as a restricted form of *generators* (as in C#). On the other hand, symmetric coroutines are generally chosen as a low-level mechanism used to implement more advanced concurrency mechanisms (as in Modula). An other example is the Unix Standard [41] where the recommended low-level primitives for implementing lightweight processes (users threads) are *getcontext*, *setcontext*, *swapcontext* and *makecontext*. This is the terminology we have previously adopted for our coroutines [12]. However, since we are working in a purely functional framework, we shall write "functional coroutines" to avoid any confusion with other mechanisms.

More recently, Anton and Thiemann described a static type system for first-class, stackful coroutines [1] that may be used in both, symmetric and asymmetric ways. They followed Danvy's method [16] to derive definitional interpreters for several styles of coroutines from the literature (starting from reduction semantics). This work is clearly very close to our formalization, and it should help shed some light on these mechanisms. However, we should keep in mind that logical deduction systems come with their own constraints which might not be compatible with existing programming paradigms.

$$x : \Gamma, A^x \vdash \Delta; A$$

$$\frac{t : \Gamma, A^x \vdash \Delta; B}{\lambda x.t : \Gamma \vdash \Delta; A \to B}(I_\to) \qquad \frac{t : \Gamma \vdash \Delta; A \to B \qquad u : \Gamma \vdash \Delta; A}{t\ u : \Gamma \vdash \Delta; B}(E_\to)$$

$$\frac{t : \Gamma \vdash \Delta; A}{\mathbf{throw}\ \alpha\ t : \Gamma \vdash \Delta, A^\alpha; B}(W_R) \qquad \frac{t : \Gamma \vdash \Delta, A^\alpha; A}{\mathbf{catch}\ \alpha\ t : \Gamma \vdash \Delta; A}(C_R)$$

Table 1: Classical Natural Deduction

2 Dependency relations

Parigot's original CND is a deduction system for the second-order classical logic. Since we are mainly interested here in the computational content of untyped terms, we shall simply recall the restriction in the propositional framework corresponding to classical logic with the implication as only connective (in Table 1). We refer the reader to [9, 12] for the full treatment of primitive conjunction, disjunction and quantifiers (including the proof that the restricted system is sound and complete for CD).

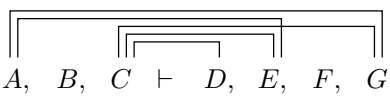
Remark. We actually work with a minor variant of the original $\lambda\mu$-calculus, called the λ_{ct}-calculus, with a primitive catch/throw mechanism [10]. These primitives are however easily definable in the $\lambda\mu$-calculus as $\mathbf{catch}\ \alpha\ t \equiv \mu\alpha[\alpha]t$ and $\mathbf{throw}\ \alpha\ t \equiv \mu\delta[\alpha]t$ where δ is a name which does not occur in t.

Since Parigot's CND is multiple-conclusioned sequent calculus, it is possible to apply so-called *Dragalin restriction* to obtain a sound and complete system for CD. This restriction requires that the succedent of the premise of the introduction rule for implication have only one formula:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash \Delta, A \to B}$$

Unfortunately, the Dragalin restriction is not stable under proof-reduction. However, a weaker restriction which is stable under proof-reduction, consists in allowing multiple conclusions in the premise of this rule, with the proviso that *these other conclusions do not depend on* A. These dependencies between occurrences of hypotheses and occurrences of conclusions in a sequent are defined by induction on the derivation.

Example. Consider a derived sequent $A, B, C \vdash D, E, F, G$ with the following dependencies:



Using named hypotheses $A^x, B^y, C^z \vdash D, E, F, G$, this annotated sequent may be represented as:

$$A^x, B^y, C^z \vdash \{z\} : D, \{x, z\} : E, \{\} : F, \{x, z\} : G$$

Let us assume now that t is the proof-term corresponding to the above derivation, i.e. we have derived in CND the following typing judgment:

$$t : A^x, B^y, C^z \vdash D^\alpha, E^\beta, F^\gamma; G$$

then we could also obtain the same dependencies directly from t, by computing sets of variables used by the various coroutines (where $[]$ refers to the distinguished conclusion), and we would get:

- $\mathcal{S}_\alpha(t) = \{z\}$
- $\mathcal{S}_\beta(t) = \{x, z\}$
- $\mathcal{S}_\gamma(t) = \{\}$
- $\mathcal{S}_{[]}(t) = \{x, z\}$

There is thus no need to actually annotate sequents with dependency relations: the relevant information is already present inside the proof-term. Let us recall how these sets are defined [12].

Definition 1. *Given a term t, for any free μ-variable δ of t, the sets of variables $\mathcal{S}_\delta(v)$ and $\mathcal{S}_{[]}(u)$ are defined inductively as follows:*

- $\mathcal{S}_{[]}(x) = \{x\}$
  $\mathcal{S}_\delta(x) = \emptyset$
- $\mathcal{S}_{[]}(\lambda x.u) = \mathcal{S}_{[]}(u) \backslash \{x\}$
  $\mathcal{S}_\delta(\lambda x.u) = \mathcal{S}_\delta(u) \backslash \{x\}$
- $\mathcal{S}_{[]}(u\ v) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_{[]}(v)$
  $\mathcal{S}_\delta(u\ v) = \mathcal{S}_\delta(u) \cup \mathcal{S}_\delta(v)$
- $\mathcal{S}_{[]}(\mathbf{catch}\ \alpha\ u) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_\alpha(u)$
  $\mathcal{S}_\delta(\mathbf{catch}\ \alpha\ u) = \mathcal{S}_\delta(u)$
- $\mathcal{S}_{[]}(\mathbf{throw}\ \alpha\ u) = \emptyset$
  $\mathcal{S}_\alpha(\mathbf{throw}\ \alpha\ u) = \mathcal{S}_\alpha(u) \cup \mathcal{S}_{[]}(u)$
  $\mathcal{S}_\delta(\mathbf{throw}\ \alpha\ u) = \mathcal{S}_\delta(u)$ *for any* $\delta \neq \alpha$

Definition 2. *A term t is* **safe** *if and only if for any subterm of t which has the form $\lambda x.u$, for any free μ-variable δ of u, $x \notin \mathcal{S}_\delta(u)$.*

Example. The term $\lambda x.\mathbf{catch}\ \alpha\ \lambda y.\mathbf{throw}\ \alpha\ x$ is safe, since x was declared before **catch** α and x is thus visible in **throw** $\alpha\ x$. On the other hand, $\lambda x.\mathbf{catch}\ \alpha\ \lambda y.\mathbf{throw}\ \alpha\ y$ is not safe, because y is not visible in **throw** $\alpha\ y$. More generally, for any α, a term of the form $\lambda y.\mathbf{throw}\ \alpha\ y$ is the reification of α as a first-class continuation and such a term is never safe. This can also be understood at the type level since the typing judgment of such a term is the law of excluded middle $\vdash A^{\alpha}; \neg A$.

Remark. You can thus decide *a posteriori* if a proof in CND is valid in CD simply by checking if the (untyped) proof-term is safe.

2.1 Safety revisited

In the conventional λ-calculus, there are two standard algorithms to decide whether a term is closed: either you build inductively the set of free variables and then check that it is empty, or you define a recursive function which takes as argument the set of declared variables, and check that each variable has been declared.

Similarly, for the $\lambda\mu$-calculus there are two ways of defining safety: the previous definition refined the standard notion of free variable (by defining a set per free μ-variable). In the following definition, *Safe* takes as arguments the sets of visible variables for each coroutine, and then decides for each variable, if the variable is visible in the current coroutine. For a closed term, *Safe* is called with $\mathcal{V}, \mathcal{V}_\mu$ both empty.

Definition 3. *The property* $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$ *by induction on* t *as follows:*

$$\begin{array}{rcl}
Safe^{\mathcal{V},\mathcal{V}_\mu}(x) & = & x \in \mathcal{V} \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(t\ u) & = & Safe^{\mathcal{V},\mathcal{V}_\mu}(t) \wedge Safe^{\mathcal{V},\mathcal{V}_\mu}(u) \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(\lambda x.t) & = & Safe^{(x::\mathcal{V}),\mathcal{V}_\mu}(t) \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(\mathbf{catch}\ \alpha\ t) & = & Safe^{\mathcal{V},(\alpha\mapsto\mathcal{V};\mathcal{V}_\mu)}(t) \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(\mathbf{throw}\ \alpha\ t) & = & Safe^{\mathcal{V}_\mu(\alpha),\mathcal{V}_\mu}(t)
\end{array}$$

where:

- $\mathcal{V}$ *is a list of variables*
- $\mathcal{V}_\mu$ *maps* μ*-variables onto lists of variables*

Remark. This definition can also be seen as the reformulation, at the level of proof-terms, of the "top-down" definition of the restriction of CND from [6] which was introduced in the framework of proof search.

As expected, we can show that the above two definitions of safety are equivalent. More precisely, the following propositions are provable.

Proposition 4. *For any term t and any mapping $\mathcal{V}_\mu$ such that $FV_\mu(t) \subseteq dom(\mathcal{V}_\mu)$, we have: $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$ implies $\mathcal{S}_{[]}(t) \subseteq \mathcal{V}$ and $\mathcal{S}_\delta(t) \subseteq \mathcal{V}_\mu(\delta)$ for any $\delta \in dom(\mathcal{V}_\mu)$ and t is safe.*

Proposition 5. *For any safe term t, for any set $\mathcal{V}$ such that $\mathcal{S}_{[]}(t) \subseteq \mathcal{V}$, for any mapping $\mathcal{V}_\mu$ such that $FV_\mu(t) \subseteq dom(\mathcal{V}_\mu)$ and $\mathcal{S}_\delta(t) \subseteq \mathcal{V}_\mu(\delta)$ for any $\delta \in FV_\mu(t)$, we have $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$.*

3 Abstract machines

In this section, we recall Groote's abstract machine for the $\lambda\mu$-calculus [18], then we present the modified machine for safe terms and we prove its correctness. But before we describe the abstract machines, we need to move to a syntax using *de Bruijn* indices, and to adapt the definition of safety.

3.1 Safe $\lambda_{\mathbf{ct}}$-terms

We rely on *de Bruijn* indices for both kind of variables (the regular variables and the μ-variables) but they correspond to different name spaces. Let us now call *vector* a list of indices (i.e. natural numbers), and *table* a list of vectors. The definition of *Safe* given for named terms can be rephrased for *de Bruijn* terms as follows. For a closed term, *Safe* is called with $\mathcal{I}$, $\mathcal{I}_\mu$ both empty and $n = 0$.

Notation 6. **We write $\backquote{g}\backquote$ for the term corresponding to variable with index g. The rest of the syntax is standard for de Bruijn terms.*

Definition 7. *Given t: term, $\mathcal{I}$: vector, $\mathcal{I}_\mu$: table and n: nat, the property $Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t)$ is defined inductively by the following rules:*

$$\frac{n - g = k \qquad k \in \mathcal{I}}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\backquote{g}\backquote)}$$

$$\frac{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t) \qquad Safe_n^{\mathcal{I},\mathcal{I}_\mu}(u)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t\ u)}$$

$$\frac{Safe_{Sn}^{(Sn::\mathcal{I}),\mathcal{I}_\mu}(t)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\lambda t)}$$

$$\frac{Safe_n^{\mathcal{I},(\mathcal{I}::\mathcal{I}_\mu)}(t)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\mathbf{catch}\ t)}$$

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad Safe_n^{\mathcal{I}',\mathcal{I}_\mu}(t)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\mathbf{throw}\ \alpha\ t)}$$

Remark. Note that n is used to count occurrences of λ from the root of the term (seen as a tree), and such a number clearly uniquely determines a λ on a branch. Since they are the numbers stored in $\mathcal{I}$, $\mathcal{I}_\mu$, a difference is computed for the base case since *de Bruijn* indices count λ beginning with the leaf.

3.2 From local indices to global indices

In the framework of environment machines, the point of using *de Bruijn* indices to represent variables is to point directly to the location of the closure in the environment (the closure which is bound to the variable). On the other hand, the intuition behind the safety property is that for each continuation, there is only a fragment of the environment which is visible (the local environment of the coroutine).

In the modified machine, these indices should point to locations in the local environment. Although the abstract syntaxes are isomorphic, it seems better to introduce a new calculus (since indices in terms have different semantics), where we can also rename **catch/throw** as **get-context/set-context** (to be consistent with the new semantics). Let us call $\lambda_{\mathbf{gs}}$-calculus the resulting calculus, and let us now define formally the translation of $\lambda_{\mathbf{gs}}$-terms onto (safe) $\lambda_{\mathbf{ct}}$-terms.

Remark. In the Coq proof assistant, it is often more convenient to represent partial functions as relations (since all functions are total in Coq we would need option types to encode partial functions). In the sequel, we call "functional" or "deterministic" any relation which has been proved functional.

Definition 8. *The functional relation* $\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (t) = t'$*, with* t*:* $\lambda_{\mathbf{gs}}$*-term,* t'*:* $\lambda_{\mathbf{ct}}$*-term,* $\mathcal{I}$*: vector,* $\mathcal{I}_\mu$*: table and* n*: nat, is defined inductively by the following rules:*

$$\frac{n - \mathcal{I}(l) = g}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\text{`}l\text{`}) = (\text{`}g\text{`})}$$

$$\frac{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (t) = t' \qquad \downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (u) = u'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (t\ u) = (t'\ u')}$$

$$\frac{\downarrow_{Sn}^{(Sn::\mathcal{I}),\mathcal{I}_\mu} (t) = t'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\lambda t) = (\lambda t')}$$

$$\frac{\downarrow_n^{\mathcal{I},(\mathcal{I}::\mathcal{I}_\mu)} (t) = t'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\mathbf{get\text{-}context}\ t) = (\mathbf{catch}\ t')}$$

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad \downarrow_n^{\mathcal{I}',\mathcal{I}_\mu} (t) = t'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\mathbf{set\text{-}context}\ \alpha\ t) = (\mathbf{throw}\ \alpha\ t')}$$

The shape of this definition is obviously very similar to the definition of safety. Actually, we can prove that a $\lambda_{\mathbf{ct}}$-term is safe if and only if it is the image of some $\lambda_{\mathbf{gs}}$-term by the translation.

Lemma 9. $\forall\ \mathcal{I}\ \mathcal{I}_\mu\ n\ t'$, $Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t') \quad \leftrightarrow \quad \exists t, \downarrow_n^{\mathcal{I},\mathcal{I}_\mu}(t) = t'$.

3.3 Abstract machine for $\lambda_{\mathbf{ct}}$-terms

De Groote's machine [18] is actually an extension of the well-known Krivine's abstract machine which has already been studied extensively in the literature [17]. Moreover, this abstract machine can also be mechanically derived from a contextual semantics of the $\lambda\mu$-calculus with explicit substitutions using the method developed by Biernacka and Danvy [4], and it is thus correct by construction.

3.3.1 Closure, environment, stack and state

Definition 10. *A closure is an inductively defined tuple* $[t, \mathcal{E}, \mathcal{E}_\mu]$ *with* t *: term,* $\mathcal{E}$ *: closure list,* $\mathcal{E}_\mu$ *: stack list (where a stack is a closure list).*

Definition 11. *A state is defined as a tuple* $\langle\ t,\ \mathcal{E},\ \mathcal{E}_\mu,\ \mathcal{S}\ \rangle$ *where* $[t, \mathcal{E}, \mathcal{E}_\mu]$ *is a closure and* $\mathcal{S}$ *is a stack.*

3.3.2 Evaluation rules

Definition 12. *The deterministic transition relation* $\sigma_1 \rightsquigarrow \sigma_2$*, with* σ_1*,* σ_2*: state, is defined inductively by the following rules:*

$$\frac{\mathcal{E}(k) = [t, \mathcal{E}', \mathcal{E}'_\mu]}{\langle \text{`}k\text{`}, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

$$\langle (tu)\,, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}, \mathcal{E}_\mu, [u, \mathcal{E}, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

$$\langle \lambda t, \mathcal{E}, \mathcal{E}_\mu, c :: \mathcal{S}\rangle \rightsquigarrow \langle t, (c :: \mathcal{E}), \mathcal{E}_\mu, \mathcal{S}\rangle$$

$$\langle \mathbf{catch}\ t, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}, (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

$$\frac{\mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \mathbf{throw}\ \alpha\ t, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

3.4 Abstract machine for $\lambda_{\mathbf{gs}}$-terms (with local environments)

As mentioned in the introduction, the modified abstract machine for λ_{gs}-terms is a surprisingly simple variant of de Groote's abstract machine, where a μ-variable is mapped onto a pair $\langle$*environment*, *continuation*$\rangle$ (a context) and not only a continuation. For simplicity, we still keep two distinct environments in the machine, $\mathcal{L}_\mu$ and $\mathcal{E}_\mu$ in a closure (but they have the same domain, which is the set of free μ-variables). As expected, the primitives **get-context** and **set-context** respectively capture and restore the local environment together with the continuation.

3.4.1 Closure, environment, stack and state

Definition 13. *A closure$_l$ is an inductively defined tuple $[t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu]$ where t: term, $\mathcal{L}$: closure$_l$ list, and $\mathcal{L}_\mu$: (closure$_l$ list) list, $\mathcal{E}_\mu$: stack$_l$ list (where a stack$_l$ is closure$_l$ list).*

Definition 14. *A state$_l$ is defined as a tuple $\langle\, t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S} \,\rangle$ where $[t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu]$ is a closure$_l$ and $\mathcal{S}$ is a stack$_l$.*

3.4.2 Evaluation rules

Definition 15. *The deterministic transition relation $\sigma_1 \rightsquigarrow^l \sigma_2$, with σ_1, σ_2: state$_l$, is defined inductively by the following rules:*

$$\frac{\mathcal{L}(k) = [t, \mathcal{L}', \mathcal{L}'_\mu, \mathcal{E}'_\mu]}{\langle \text{`}k\text{'}, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}', \mathcal{L}'_\mu, \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

$$\langle (tu)\,, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, [u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

$$\langle \lambda t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, c :: \mathcal{S}'\rangle \rightsquigarrow^l \langle t, (c :: \mathcal{L}), \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}'\rangle$$

$$\langle \textbf{get-context}\ t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}, (\mathcal{L} :: \mathcal{L}_\mu), (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

$$\frac{\mathcal{L}_\mu(\alpha) = \mathcal{L}' \qquad \mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{set-context}\ \alpha\ t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}', \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

4 Bisimulations

We first introduce the intermediate machine for λ_{gs}-terms, then we define two functional bi-simulations $(-)^\star$ and $(-)^\diamond$ showing that this intermediate machine:

- bi-simulates the regular machine with global indices
- bi-simulates the modified machine with local environments

4.1 Abstract machine for λ_{gs}-terms (with indirection tables)

This intermediate machine for λ_{gs}-terms works *with local indices, global environment and indirection tables*. The indirection tables are exactly the same as for the static translation of λ_{gs}-terms to safe λ_{ct}-terms. However, the translation is now performed at runtime. The lock-step simulation $(-)^\star$ shows that translating during evaluation is indeed equivalent to evaluating the translated term. The lock-step simulation $(-)^\diamond$ shows that we can "flatten away" the indirection tables and the global environment, and work only with local environments.

4.1.1 Closure, environment, stack and state

Definition 16. *A closure$_i$ is an inductively defined tuple $[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]$, with t : term, n : nat, $\mathcal{I}$: vector, $\mathcal{I}_\mu$: table, $\mathcal{E}$: closure$_i$ list, $\mathcal{E}_\mu$: stack$_i$ list (where a stack$_i$ is a closure$_i$ list).*

Definition 17. *A state$_i$ is defined as a tuple $\langle\ t,\ n,\ \mathcal{I},\ \mathcal{I}_\mu,\ \mathcal{E},\ \mathcal{E}_\mu,\ \mathcal{S}\ \rangle$ where $[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]$ is a closure$_i$ and $\mathcal{S}$ is a stack$_i$.*

4.1.2 Evaluation rules

Definition 18. *The deterministic transition relation $\sigma_1 \rightsquigarrow^i \sigma_2$, with σ_1, σ_2: state$_i$, is defined inductively by the following rules:*

$$\frac{n - \mathcal{I}(l) = g \qquad \mathcal{E}(g) = [t, n', \mathcal{I}', \mathcal{I}'_\mu, \mathcal{E}', \mathcal{E}'_\mu]}{\langle \backslash l \backslash, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n', \mathcal{I}', \mathcal{I}'_\mu, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

$$\langle (tu), n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, [u, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

$$\langle \lambda t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, c :: \mathcal{S}'\rangle \rightsquigarrow^i \langle t, (Sn), ((Sn) :: \mathcal{I}), \mathcal{I}_\mu, c :: \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle$$

$$\langle \textbf{get-context}\ t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}, (\mathcal{I} :: \mathcal{I}_\mu), \mathcal{E}, (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad \mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{set-context}\ \alpha\ t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}', \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

4.2 Lock-step simulation $(-)^\star$

Definition 19. *The functional relation $c^\star =_c c'$, with c: closure$_i$, c': closure, is defined by the following rule:*

$$\frac{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu}(t) = u \qquad \mathcal{E}^\star =_e \mathcal{E}' \qquad \mathcal{E}^\star_\mu =_k \mathcal{E}'_\mu}{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\star =_c [u, \mathcal{E}', \mathcal{E}'_\mu]}$$

where $\mathcal{E}^\star$ and $\mathcal{E}^\star_\mu$ are defined by element-wise application of *.

Definition 20. *The functional relation $\sigma^\star =_\sigma \sigma'$, with σ: state$_i$, σ': state, is defined by the following rule:*

$$\frac{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\star =_c [u, \mathcal{E}', \mathcal{E}'_\mu] \qquad \mathcal{S}^\star =_s \mathcal{S}'}{\langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle^\star =_\sigma \langle u, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}'\rangle}$$

where $\mathcal{S}^\star$ is defined by element-wise application of *.

4.2.1 Soundness

Theorem 21. $\forall\ \sigma_1\ \sigma_2\ \sigma_1',$
$\sigma_1 \rightsquigarrow^i \sigma_2\ \rightarrow\ \sigma_1^\star =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2',\ \sigma_1' \rightsquigarrow \sigma_2'\ \wedge\ \sigma_2^\star =_\sigma \sigma_2'.$

4.2.2 Completeness

Theorem 22. $\forall\ \sigma_1'\ \sigma_2'\ \sigma_1,$
$\sigma_1' \rightsquigarrow \sigma_2'\ \rightarrow\ \sigma_1^\star =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2,\ \sigma_1 \rightsquigarrow^i \sigma_2\ \wedge\ \sigma_2^\star =_\sigma \sigma_2'.$

4.3 Lock-step simulation $(-)^\diamond$

Definition 23. *The functional relation $c^\diamond =_k c'$ with c: closure$_i$, c': closure$_l$, is defined by the following rule:*

$$\frac{\mathit{flatten}\ n\ \mathcal{E}\ \mathcal{I}\ =\ \mathcal{L} \qquad \mathit{map}\ (\mathit{flatten}\ n\ \mathcal{E})\ \mathcal{I}_\mu\ =\ \mathcal{L}_\mu \qquad \mathcal{E}_\mu^\diamond =_k \mathcal{E}_\mu'}{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\diamond =_c [t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu']}$$

where $\mathcal{S}^\diamond$ and $\mathcal{E}_\mu^\diamond$ are defined by element-wise application of $^\diamond$, and the functional relation ($\mathit{flatten}\ n\ \mathcal{E}\ \mathcal{I} = \mathcal{L}$) is defined inductively by the following rules:

$$\mathit{flatten}\ n\ \mathcal{E}\ \mathit{nil}\ =\ \mathit{nil} \qquad \frac{\mathcal{E}(n-k) = c \qquad c^\diamond =_c c' \qquad \mathit{flatten}\ n\ \mathcal{E}\ \mathcal{I}\ =\ \mathcal{L}}{\mathit{flatten}\ n\ \mathcal{E}\ (k :: \mathcal{I})\ =\ (c' :: \mathcal{L})}$$

Definition 24. *The functional relation $\sigma^\diamond =_\sigma \sigma'$, with σ : state$_i$, σ': state$_l$, is defined by the following rule:*

$$\frac{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\diamond =_c [u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu'] \qquad \mathcal{S}^\diamond =_s \mathcal{S}'}{\langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle^\diamond =_\sigma \langle u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu', \mathcal{S}'\rangle}$$

4.3.1 Soundness

Theorem 25. $\forall\ \sigma_1\ \sigma_2\ \sigma_1',$
$\sigma_1 \rightsquigarrow^i \sigma_2\ \rightarrow\ \sigma_1^\diamond =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2',\ \sigma_1' \rightsquigarrow^l \sigma_2'\ \wedge\ \sigma_2^\diamond =_\sigma \sigma_2'.$

4.3.2 Completeness

Theorem 26. $\forall\ \sigma_1'\ \sigma_2'\ \sigma_1,$
$\sigma_1' \rightsquigarrow^l \sigma_2'\ \rightarrow\ \sigma_1^\diamond =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2,\ \sigma_1 \rightsquigarrow^i \sigma_2\ \wedge\ \sigma_2^\diamond =_\sigma \sigma_2'.$

5 Conclusion and future work

We have defined and formally proved the correctness of an abstract machine which provides a direct computational interpretation of the Constant Domain logic. However, as mentioned in the introduction, this work is a stepping stone towards a computational interpretation of duality in intuitionistic logic. Starting from the reduction semantics of proof-terms of bi-intuitionistic logic (subtractive

logic) [12], it should be possible to extend the modified machine to account for first-class coroutines.

These results should then be compared with other related works, such as Curien and Herbelin's pioneering article on the duality of computation [13], or more recently, Bellin and Menti's work on the π-calculus and co-intuitionistic logic [3] and Kimura and Tatsuta's Dual Calculus [28].

Acknowledgments. I would like to thank Nuria Brede for numerous discussions on the "safe" $\lambda\mu$-calculus and useful comments on earlier versions of this work. I am also very grateful to Olivier Danvy for proof-reading this article, and for giving me the opportunity to present these results at WoC 2015.

References

[1] K. Anton and P. Thiemann. Towards deriving type systems and implementations for coroutines. In Kazunori Ueda, editor, *Programming Languages and Systems – 8th Asian Symposium, APLAS 2010*, volume 6461 of *LNCS*, pages 63–79, Shanghai, China, 2010. Springer. 4

[2] F. Barbanera and S. Berardi. Extracting Constructive Content from Classical Logic via Control-like Reductions. In *LNCS*, volume 662, pages 47–59. Springer-Verlag, 1994. 3

[3] G. Bellin and A. Menti. On the π-calculus and Co-intuitionistic Logic. Notes on Logic for Concurrency and λP Systems. *Fundamenta Informaticae*, 130(1):21–65, January 2014. 14

[4] M. Biernacka and O. Danvy. A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines. *Theoretical Computer Science*, 375, 2007. 10

[5] D. Biernacki, O. Danvy, and C. Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006. 4

[6] N. Brede. $\lambda\mu$PRL - A Proof Refinement Calculus for Classical Reasoning in Computational Type Theory. Master's thesis, University of Potsdam, 2009. 7

[7] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2(1):50–63, 1973. 3

[8] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963. 3

[9] T. Crolard. Extension de l'Isomorphisme de Curry-Howard au Traitement des Exceptions (application d'une étude de la dualité en logique intuitionniste). Thèse de Doctorat. Université Paris 7, 1996. 1, 2, 5

[10] T. Crolard. A confluent lambda-calculus with a catch/throw mechanism. *Journal of Functional Programming*, 9(6):625–647, 1999. 5

[11] T. Crolard. Subtractive Logic. *Theoretical Computer Science*, 254(1–2):151–185, 2001. 1

[12] T. Crolard. A Formulæ-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation*, 14(4):529–570, 2004. 1, 4, 5, 6, 14

[13] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 233–243, New York, USA, 2000. ACM Press. 14

[14] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. Academic Press, 1972. 4

[15] O.-J. Dahl and K. Nygaard. SIMULA: an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, 1966. 3

[16] O. Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 131–142, New York, USA, 2008. ACM Press. 4

[17] O. Danvy, editor. Special Issue on the Krivine Machine. *Higher-Order and Symbolic Computation*, 20(3), 2007. 10

[18] P. de Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structure in Computer Science*, 8:637–669, 1998. 2, 8, 10

[19] P. de Groote. Strong normalization of classical natural deduction with disjunction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *LNCS*, pages 182–196. Springer, 2001. 3

[20] A. L. de Moura and R. Ierusalimschy. Revisiting Coroutines. MCC 15/04, PUC-Rio, Rio de Janeiro, RJ, June 2004. 4

[21] A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004. 3

[22] D. P. Friedman, C. T. Haynes, and M. Wand. Obtaining Coroutines with Continuations. *Journal of Computer Languages*, 11(3/4):143–153, 1986. 3

[23] S. Görnemann. A logic stronger than intuitionism. *The Journal of Symbolic Logic*, 36:249–261, 1971. 1

[24] T. G. Griffin. A formulæ-as-types notion of control. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Langages*, pages 47–58, 1990. 3

[25] A. Grzegorczyk. A philosophically plausible formal interpretation of intuitionistic logic. *Nederl. Akad. Wet., Proc., Ser. A*, 67:596–601, 1964. 1

[26] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. 3

[27] R. Kashima. Cut-Elimination for the intermediate logic CD. Research Report on Information Sciences C100, Institute of Technology, Tokyo, 1991. 1

[28] D. Kimura and M. Tatsuta. Dual calculus with inductive and coinductive types. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil*, volume 5595 of *LNCS*, pages 224–238. Springer, 2009. 14

[29] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 2nd edition edition, 1973. 3

[30] J.-L. Krivine. Classical logic, storage operators and second order λ-calculus. *Ann. of Pure and Appl. Logic*, 68:53–78, 1994. 3

[31] E. G. K. Lopez-Escobar. A Second Paper "On the Interpolation Theorem for the Logic of Constant Domains". *The Journal of Symbolic Logic*, 48(3):595–599, 1983. 1

[32] C. D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1980. 3

[33] G. Mints, G. Olkhovikov, and A. Urquhart. Failure of interpolation in constant domain intuitionistic logic. *The Journal of Symbolic Logic*, 78:937–950, 9 2013. 1

[34] C. R. Murthy. Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, Department of Computer Science, 1991. 3

[35] M. Parigot. Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*, 1993. 1, 3

[36] D. Pym, E. Ritter, and L. Wallen. On the intuitionistic force of classical search. *Theoretical Computer Science*, 232(1-2):299–333, 2000. 3

[37] C. Rauszer. An algebraic and Kripke-style approach to a certain extension of intuitionistic logic. In *Dissertationes Mathematicae*, volume 167. Institut Mathématique de l'Académie Polonaise des Sciences, 1980. 1

[38] N. J. Rehof and M. H. Sørensen. The λ_Δ-calculus. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 516–542. Springer-Verlag, 1994. 3

[39] J. H. Reppy. First-class synchronous operations. In *Proceedings of the International Workshop on Theory and Practice of Parallel Programming*, volume 907 of *LNCS*, pages 235–252, London, UK, 1995. Springer-Verlag. 3

[40] T. Streicher and B. Reus. Classical Logic, Continuation Semantics and Abstract Machines. *Journal of Functional Programming*, 8(6):543–572, 1998. 2

[41] The Open Group. The Single UNIX Specification, Version 2, 1997. 4

[42] N. Wirth and J. Mincer-Daszkiewicz. *Modula-2*. ETH Zurich, Schweiz, 1980. 3