A verified abstract machine for functional coroutines

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

Functional coroutines are a restricted form of control mechanism, where each coroutine is represented with both a continuation and an 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 for functional coroutines and we prove its correctness. Therefore, this abstract machine also provides a direct computational interpretation of the Constant Domain logic.

1 Introduction

The *Constant Domain* logic (CD) is a well-known intermediate logic due to Grzegorczyk [30] 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 [37], and it took more than three decades to prove that the interpolation theorem does not hold either [40]. However, CD is unavoidable when the object of study is *duality in intuitionistic logic*. Indeed, consider the following schema (called either D [30] or DIS [50]), 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 [50] or subtractive logic [12]), 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 [25] (and also that the disjunction and existence properties hold, so CD is still a constructive logic). Moreover, Rauszer proved that bi-intuitionistic logic is conservative over CD [50] (Section 3, p. 56), which means that the theorems of bi-intuitionistic logic with no occurence of subtraction are exactly the theorems of CD. 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 and Shimura [32, 53] as a restriction of Gentzen's sequent calculus LK based on dependency relations. Independently, we described [10] a similar restriction using Parigot's classical natural deduction [44] 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 [13]. 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 [28] (a very similar machine was defined independently by Streicher and Reus [55]). Then we define a new variant of this machine dedicated to the execution of safe terms which works exactly as hinted above (let us call it the *coroutine machine*). 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 is more convenient to encode variables as de Bruijn indices (in particular for correctness proofs). 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 arbitrary 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 coroutine machine which is two-fold. We first introduce an intermediate machine which works with *local indices, global environment and indirection tables*, then we show that this intermediate machine:

- is simulated by de Groote's machine,
- is simulated by the coroutine machine.

We prove that both simulations are sound and complete, and as a consequence, we obtain the correctness of the coroutine machine with respect to de Groote's machine.

The plan of the paper is the following. In Section 2, we first recall the notion of safety [10], and then we present a simpler (but equivalent) definition of safety which is more convenient for correctness proofs. In Section 3, we present our variant of de Groote's machine and the coroutine machine. Finally, in Section 4, we detail the proof of correctness: we describe the intermediate machine and the two simulations together with their properties (all the proofs were mechanically checked with the Coq proof assistant, and the formalization is available in the companion technical report [14]).

1.1 Related work

Computational interpretation of classical logic

Since Griffin's pioneering work [27], the extension of the well-known formulas-as-types paradigm to classical logic has been widely investigated for instance by Murthy [43], Barbanera and Berardi [3], Rehof and Sørensen [51], de Groote [29], and Krivine [35]. We shall consider here Parigot's $\lambda\mu$-calculus mainly because it is confluent and strongly normalizing in the second order framework [44]. 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 [48] and de Groote [29].

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 [58, 54]. 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 [23] or to provide simple and elegant implementations of light-weight processes (or threads) [20]. This approach has also been applied in Standard ML of New Jersey [52] using the typed counterpart of Scheme's **call/cc** [31]. 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 [9] who introduced it to describe the interaction between a lexer and a parser inside a compiler. They were also used by Knuth [34] (Section 1.4.2, p. 193) who saw them as a mechanism that generalizes subroutines (procedures without parameters). Coroutines first appeared in a mainstream language in Simula-67 [17] and a formal framework for proving the correctness of simple Simula programs containing coroutines has even been developed [8]. Coroutine mechanisms were later introduced in several programming languages, for instance in Modula-2 [59], and more recently in the functional language Lua [41, 42].

Marlin's thesis [38], which is cited as a reference for coroutines implementations [41], 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 described as follows [41]:

- *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 within 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* [16], 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 the other coroutine when it encounters a leaf. The

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

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

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

Table 1: Classical Natural Deduction

elegance of this algorithm lies in the fact that each coroutine uses its own stack, which permits for two simple recursive tree traversals.

Although the first occurrence of the same-fringe problem in the litterature seems indeed to be an illustration of a coroutine mechanism [47], researchers did not agree on whether coroutines were really required to solve this problem. In fact, several "iterative" solutions were then proposed for instance by Greussay [26], Anderson [1] and McCarthy [39]. In fact, a variety of inter-derivable solutions of this problem exist that do not solely rely on coroutines [6]. However, since we are also interested in program logics, it is relevant to quote McCarthy's conclusion about his own solution: "*A program with only assignments and goto's may have the most easily modified control structure. Of course, elegance, understandability and a control logic admitting straightforward proofs of correctness are also virtues*".

More recently, Anton and Thiemann described a static type system for first-class, stackful coroutines [2] that may be used in both, symmetric and asymmetric ways. They followed Danvy's method [18] to derive definitional interpreters for several styles of coroutines from the literature (starting from reduction semantics for Lua). 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 fully compatible with existing programming paradigms: nobody knows to what extent what Griffin did for continuations [22] can be done for coroutines.

Remark. 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 such example is the Unix Standard [56] 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 [13]. However, since we are working in a purely functional framework, we shall write "functional coroutines" to avoid any confusion with other mechanisms.

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 [10, 13] for the full treatment of primitive conjunction, disjunction and quantifiers (including the proof that the restricted system is sound and complete for CD).

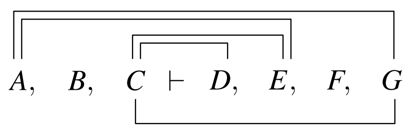
Remark. We actually work with a minor variant of the original $\lambda\mu$-calculus, called the λ_{ct}-calculus, with a primitive catch/throw mechanism [11]. These primitives are however easily definable in the $\lambda\mu$-calculus as **catch** $\alpha\ t \equiv \mu\alpha[\alpha]t$ and **throw** $\alpha\ t \equiv \mu\delta[\alpha]t$ 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:

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

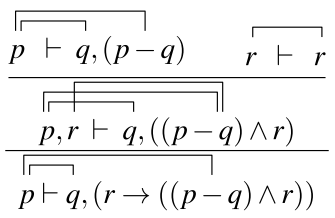
then we could also obtain the same dependencies directly from t, by computing sets of variables used by the various coroutines (where [] refers to the distinguished conclusion), and we would get:

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

Remark. Deduction systems which rely on the Dragalin restriction usually do not enjoy the cut elimination property: there are some derivable sequents for which no cut-free proof exists. Pinto and Uustalu [45] recently presented such a counter-example (which is also mentionned by Goré and Postniece [24]) for Rauszer's sequent calculus for bi-intuitionistic logic [49]. They show that there is no cut-free proof of the following sequent in Rauszer calculus:

$$p \vdash q, (r \to ((p - q) \wedge r))$$

There is however a cut-free proof of this sequent in various extended sequent calculi [24, 45, 46] and, as expected, there is also a cut-free proof in the dependency-based system for bi-intuitionistic logic [13]:


$p \vdash q, (p - q)$
$r \vdash r$
$p, r \vdash q, ((p - q) \wedge r)$
$p \vdash q, (r \to ((p - q) \wedge r))$

The above presentation assumes that dependencies are explicitly displayed in derivations. In fact, there is no need to actually annotate sequents with dependency relations: the relevant information is already present inside the proof term. Let us recall how these dependencies can be extracted [13]. In the following definition $\mathscr{S}_\delta(t)$ corresponds to the set of variables of t which are *used* by coroutine δ, whereas $\mathscr{S}_{[]}(t)$ corresponds to the set of variables of t which are *used* by the "current" coroutine.

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

- $\mathscr{S}_{[]}(x) = \{x\}$
  $\mathscr{S}_\delta(x) = \emptyset$
- $\mathscr{S}_{[]}(\lambda x.u) = \mathscr{S}_{[]}(u) \backslash \{x\}$
  $\mathscr{S}_\delta(\lambda x.u) = \mathscr{S}_\delta(u) \backslash \{x\}$
- $\mathscr{S}_{[]}(u\, v) = \mathscr{S}_{[]}(u) \cup \mathscr{S}_{[]}(v)$
  $\mathscr{S}_\delta(u\, v) = \mathscr{S}_\delta(u) \cup \mathscr{S}_\delta(v)$
- $\mathscr{S}_{[]}(\textbf{catch}\ \alpha\ u) = \mathscr{S}_{[]}(u) \cup \mathscr{S}_\alpha(u)$
  $\mathscr{S}_\delta(\textbf{catch}\ \alpha\ u) = \mathscr{S}_\delta(u)$
- $\mathscr{S}_{[]}(\textbf{throw}\ \alpha\ u) = \emptyset$
  $\mathscr{S}_\alpha(\textbf{throw}\ \alpha\ u) = \mathscr{S}_\alpha(u) \cup \mathscr{S}_{[]}(u)$
  $\mathscr{S}_\delta(\textbf{throw}\ \alpha\ u) = \mathscr{S}_\delta(u)$ *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, $x \notin \mathscr{S}_\delta(u)$.*

Example. The term $\lambda x.$**catch** α $\lambda y.$**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, $\lambda x.$**catch** α $\lambda y.$**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 $\lambda y.$**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.

Example. Here is a derivation of schema DIS in CND (using a primitive disjunction):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{u : \forall x(A \vee B)^u \vdash \forall x(A \vee B)}{u : \forall x(A \vee B)^u \vdash A \vee B} \qquad A^a \vdash A \qquad \dfrac{\dfrac{\dfrac{b : B^b \vdash B}{\textbf{inr}\ b : B^b \vdash (A \vee B)}}{\textbf{inr}\ b : B^b \vdash \forall x(A \vee B)}}{\textbf{throw}\ \alpha\ (\textbf{inr}\ b) : \forall x(A \vee B)^u \vdash \forall x(A \vee B)^\alpha ; A}}{\textbf{case}\ u\ \textbf{of}\ \ \textbf{inl}\ a \to a \mid \textbf{inr}\ b \to \textbf{throw}\ \alpha\ (\textbf{inr}\ b) : \forall x(A \vee B)^u \vdash \forall x(A \vee B)^\alpha ; A}}{\textbf{case}\ u\ \textbf{of}\ \ \textbf{inl}\ a \to a \mid \textbf{inr}\ b \to \textbf{throw}\ \alpha\ (\textbf{inr}\ b) : \forall x(A \vee B)^u \vdash \forall x(A \vee B)^\alpha ; \forall x A}}{\textbf{inl}\ (\textbf{case}\ u\ \textbf{of}\ \ \textbf{inl}\ a \to a \mid \textbf{inr}\ b \to \textbf{throw}\ \alpha\ (\textbf{inr}\ b)) : \forall x(A \vee B)^u \vdash \forall x(A \vee B)^\alpha ; (\forall x A) \vee B}}{\textbf{catch}\ \alpha\ (\textbf{inl}\ (\textbf{case}\ u\ \textbf{of}\ \ \textbf{inl}\ a \to a \mid \textbf{inr}\ b \to \textbf{throw}\ \alpha\ (\textbf{inr}\ b))) : \forall x(A \vee B)^u \vdash (\forall x A) \vee B}$$

It is possible to extend the definition of safety to the primitive disjunction and then check that this proof term is safe [10, 13]. An alternative consists in relying on the usual definition of disjunction in Heyting arithmetic:

$$A \vee B \equiv \exists x : int(x = 0 \Rightarrow A \wedge x \neq 0 \Rightarrow B)$$

In this case, the proof term would be:

$$\textbf{catch}\ \alpha\ (\textbf{inl}\ (\textbf{if}\ \pi_0(u) = 0\ \textbf{then}\ \pi_1(u)\ \textbf{else}\ \textbf{throw}\ \alpha\ (\textbf{inr}\ \pi_2(u))))$$

One can then check that this proof term is indeed safe (where integers, tuples and projections can be encoded in the pure λ-calculus). Note however that the standard second-order encoding of the disjunction in the pure λ-calculus [22] does not work for this derivation, since the second branch of the **case** statement is not safe when encoded as a λ-abstraction.

Remark. As already noted in Troelstra's monograph [57] (Section 1.11.3, p. 92), adding the DIS-schema to Heyting Arithmetic yields classical arithmetic (Peano's Arithmetic). To be more specific, if atomic formulas are decidable (which is the case in HA), one can prove using DIS that *any* formula is decidable. Fortunately, there is another way to combine DIS with intuitionistic arithmetic which does not suffer from this drawback. Indeed, Leivant introduced system $\mathbf{IT}(\mathbb{N})$ [36] as an intuitionistic first-order theory where "being a natural number" is expressed using a unary predicate and where quantifiers over natural numbers need to be relativized. One can check using standard Kripke semantics that $\mathbf{IT}(\mathbb{N})$ can be extended with (non-relativized) DIS and that the resulting system is still conservative over Heyting Arithmetic (when restricted to relativized formulas).

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 (as a synthesized attribute) and then check that it is empty, or you define a recursive function which takes as argument the set of declared variables (as an inherited attribute), and checks 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 $\mathscr{V}, \mathscr{V}_\mu$ both empty.

Definition 3. *The property $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$ is defined 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 [7] 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 [28], 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.

Remark. Note that it is also possible to start with an abstract machine for $\lambda\mu$-terms with names (instead of *de Bruijn* indices) as proposed by Streicher and Reus [55], and then define a variant of this machine tailored for safe $\lambda\mu$-terms. The main advantage of this approach would be to keep a single syntax (since safety is just a predicate and no compilation is required). However, this apparent simplicity is misleading: it actually more difficult to formally prove the correctness of such a machine (since we have to deal with bound variables and α-conversion).

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 (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 \`g\` for the term consisting only of the variable with index g (this is just an explicit notation for the constructor that takes an index and builds a term). The rest of the syntax is standard for de Bruijn terms. In particular, since* **catch** *is also a binder, it takes only a term as argument in de Bruijn notation (this is similar to the λ-abstraction).*

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

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

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

$$\frac{Safe_{n+1}^{(n+1::\mathscr{I}),\mathscr{I}_\mu}(t)}{Safe_n^{\mathscr{I},\mathscr{I}_\mu}(\lambda t)}$$

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

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

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 $\mathscr{I}$, $\mathscr{I}_\mu$, 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, de Bruijn indices are used to represent variables in order to point directly to their denotation 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 is more convenient 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 $\downarrow_n^{\mathscr{I},\mathscr{I}_\mu}(t)=t'$, with t: $\lambda_{\mathbf{gs}}$-term, t': $\lambda_{\mathbf{ct}}$-term, $\mathscr{I}$: vector, $\mathscr{I}_\mu$: table and n: nat, is defined inductively by the following rules:*

$$\frac{n-\mathscr{I}(l)=g}{\downarrow_n^{\mathscr{I},\mathscr{I}_\mu}(\grave{}l\grave{})=(\grave{}g\grave{})}$$

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

$$\frac{\downarrow_{n+1}^{(n+1::\mathscr{I}),\mathscr{I}_\mu}(t) = t'}{\downarrow_n^{\mathscr{I},\mathscr{I}_\mu}(\lambda t) = (\lambda t')}$$

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

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

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. $\forall\ \mathscr{I}\ \mathscr{I}_\mu\ n\ t'$*,* $\mathit{Safe}_n^{\mathscr{I},\mathscr{I}_\mu}(t') \quad \leftrightarrow \quad \exists t, \downarrow_n^{\mathscr{I},\mathscr{I}_\mu}(t) = t'$.

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

De Groote's machine [28] is an extension of the well-known Krivine's abstract machine (K-machine) which has already been studied extensively in the literature [19]. Moreover, this abstract machine has also been derived mechanically from a contextual semantics of the $\lambda\mu$-calculus with explicit substitutions using the method developed by Biernacka and Danvy [5], and it is thus correct by construction. The $\mathrm{K_{ct}}$-machine we describe below is a variant de Groote's machine tailored for λ_{ct}-terms.

3.3.1 Closure, environment, stack and state of the $\mathbf{K_{ct}}$-machine

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

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

3.3.2 Evaluation rules for the $\mathbf{K_{ct}}$-machine

Definition 12. *The deterministic transition relation* $\sigma_1 \leadsto \sigma_2$*, with* σ_1*,* σ_2*: state, is defined inductively by the following rules (where* $\mathscr{E}(k)$ *is the k-th closure in* $\mathscr{E}$ *and* $\mathscr{E}_\mu(\alpha)$ *is the* α*-th stack in* $\mathscr{E}_\mu$*):*

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

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

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

$$\langle \textbf{catch}\, t,\mathscr{E},\mathscr{E}_\mu,\mathscr{S}\rangle \leadsto \langle t,\mathscr{E},(\mathscr{S} :: \mathscr{E}_\mu),\mathscr{S}\rangle$$

$$\frac{\mathscr{E}_\mu(\alpha) = \mathscr{S}'}{\langle \textbf{throw}\ \alpha\, t,\mathscr{E},\mathscr{E}_\mu,\mathscr{S}\rangle \leadsto \langle t,\mathscr{E},\mathscr{E}_\mu,\mathscr{S}'\rangle}$$

3.4 The $\mathrm{K_{gs}}$-machine for λ_{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. 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 of the $\mathrm{K_{gs}}$-machine

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

Remark. For simplicity, we keep two distinct mappings in a closure, $\mathscr{L}_\mu$ and $\mathscr{E}_\mu$, but they have the same domain, which is the set of free μ-variables. A μ-variable is then mapped onto a pair $\langle$*environment*, *continuation*$\rangle$ as expected: the environment is obtained from $\mathscr{L}_\mu$ and the continuation is obtained from $\mathscr{E}_\mu$.

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

3.4.2 Evaluation rules for the $\mathrm{K_{gs}}$-machine

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:*

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

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

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

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

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

4 Bisimulations

We first introduce the intermediate machine for λ_{gs}-terms, called the $\mathrm{K^{it}_{gs}}$-machine, and we define two simulations $(-)^\star$ and $(-)^\diamond$ showing that this $\mathrm{K^{it}_{gs}}$-machine is simulated by both the $\mathrm{K_{ct}}$-machine and the $\mathrm{K_{gs}}$-machine. Moreover, we shall prove that both simulations are sound and complete.

4.1 The K_{gs}^{it}-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 of the K_{gs}^{it}-machine

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

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

4.1.2 Evaluation rules for the K_{gs}^{it}-machine

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:*

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

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

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

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

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

4.2 Lock-step simulation $(-)^\star$ of the K_{gs}^{it}-machine by the K_{ct}-machine

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

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

where $\mathscr{E}^\star$ and $\mathscr{E}_\mu^\star$ 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:*

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

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

4.2.1 Soundness of simulation $(-)^{\star}$

The following theorem states that the $\mathrm{K}^{\mathrm{it}}_{\mathrm{gs}}$-machine is sound with respect to the K_{ct}-machine.

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

4.2.2 Completeness of simulation $(-)^{\star}$

The following theorem states that the $\mathrm{K}^{\mathrm{it}}_{\mathrm{gs}}$-machine is complete with respect to the K_{ct}-machine.

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

Remark. Since simulation $(-)^{\star}$ is sound and complete, and since both the $\mathrm{K}^{\mathrm{it}}_{\mathrm{gs}}$-machine and K_{gs}-machine evaluate the same λ_{gs}-terms, simulation $(-)^{\star}$ is actually a bi-simulation.

4.3 Lock-step simulation $(-)^{\diamond}$ of the $\mathrm{K}^{\mathrm{it}}_{\mathrm{gs}}$-machine by the K_{gs}-machine

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

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

where $\mathscr{S}^{\diamond}$ and $\mathscr{E}_{\mu}^{\diamond}$ are defined by element-wise application of $^{\diamond}$, and flatten is a functional relation inductively defined by the following rules:

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

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

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

4.3.1 Soundness of simulation $(-)^{\diamond}$

The following theorem states that the $\mathrm{K}^{\mathrm{it}}_{\mathrm{gs}}$-machine is sound with respect to the K_{gs}-machine.

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

4.3.2 Completeness of simulation $(-)^{\diamond}$

The following theorem states that the $\mathrm{K}^{\mathrm{it}}_{\mathrm{gs}}$-machine is complete with respect to the K_{gs}-machine.

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

Remark. Since simulation $(-)^{\diamond}$ is sound and complete, $(-)^{\diamond}$ is a bi-simulation when the initial states of the K_{ct}-machine are restricted to safe λ_{ct}-terms (by Lemma 9). However, the K_{ct}-machine can also evaluate arbitrary λ_{ct}-terms.

4.4 Lock-step simulation of the $\mathrm{K_{gs}}$-machine by the $\mathrm{K_{ct}}$-machine

We have proved that the $\mathrm{K^{it}_{gs}}$-machine is simulated by both the $\mathrm{K_{ct}}$-machine and the $\mathrm{K_{gs}}$-machine (and that both simulations are sound and complete). These properties are illustrated by the following diagram:

$$\begin{tikzcd}
\mathrm{K_{ct}}\text{-machine} & \sigma_0^\star \arrow[r, squiggly] & \cdots \arrow[r, squiggly] & \sigma_n^\star \arrow[r, squiggly] & \sigma_{n+1}^\star \arrow[r, squiggly] & \cdots \\
\mathrm{K^{it}_{gs}}\text{-machine} & \sigma_0 \arrow[u, "\star"'] \arrow[d, "\diamond"'] \arrow[r, squiggly] & \cdots \arrow[r, squiggly] & \sigma_n \arrow[u, "\star"'] \arrow[d, "\diamond"'] \arrow[r, squiggly] & \sigma_{n+1} \arrow[u, "\star"'] \arrow[d, "\diamond"'] \arrow[r, squiggly] & \cdots \\
\mathrm{K_{gs}}\text{-machine} & \sigma_0^\diamond \arrow[r, squiggly] & \cdots \arrow[r, squiggly] & \sigma_n^\diamond \arrow[r, squiggly] & \sigma_{n+1}^\diamond \arrow[r, squiggly] & \cdots
\end{tikzcd}$$

The composition of simulation $(-)^\diamond$ and $(-)^\star$ gives us a sound and complete lock-step simulation of the $\mathrm{K_{gs}}$-machine by the $\mathrm{K_{ct}}$-machine.

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) [13], it should be possible to extend the coroutine machine to account for first-class coroutines.

These future results should then be compared with other related works, such as Curien and Herbelin's pioneering article on the duality of computation [15], or more recently, Bellin and Menti's work on the π-calculus and co-intuitionistic logic [4], Kimura and Tatsuta's Dual Calculus [33] and Eades, Stump and McCleeary's Dualized simple type theory [21].

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 and Ugo de'Liguoro for giving me the opportunity to present these results at WoC 2015, and for their work as editors of this special issue. Finally, I would like to thank anonymous referees for their very constructive comments, which helped me to substantially improve the manuscript.

References

[1] B. Anderson (1976): *The Samefringe Problem*. SIGART Bull. 60, pp. 4–4.

[2] K. Anton & P. Thiemann (2010): *Towards Deriving Type Systems and Implementations for Coroutines*. In Kazunori Ueda, editor: Programming Languages and Systems – 8th Asian Symposium, APLAS 2010, LNCS 6461, Springer, Shanghai, China, pp. 63–79, doi:10.1007/ 978-3-642-17164-2_6.

[3] F. Barbanera & S. Berardi (1994): *Extracting Constructive Content from Classical Logic via Control-like Reductions*. In: LNCS, 662, Springer-Verlag, pp. 47–59, doi:10.1.1.120.386.

[4] G. Bellin & A. Menti (2014): *On the π-calculus and Co-intuitionistic Logic. Notes on Logic for Concurrency and λP Systems*. Fundamenta Informaticae 130(1), pp. 21–65, doi:10.3233/FI-2014-981.

[5] M. Biernacka & O. Danvy (2007): *A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*. Theoretical Computer Science 375, doi:10.1016/j.tcs.2006.12.028.

[6] D. Biernacki, O. Danvy & C. Shan (2006): *On the Static and Dynamic Extents of Delimited Continuations*. Science of Computer Programming 60(3), pp. 274–297, doi:10.1016/j.scico.2006.01.002.

[7] N. Brede (2009): *$\lambda\mu$PRL - A Proof Refinement Calculus for Classical Reasoning in Computational Type Theory*. Master's thesis, University of Potsdam. Available at `http://www.cs.uni-potsdam.de/~brede`.

[8] M. Clint (1973): *Program proving: Coroutines*. Acta Informatica 2(1), pp. 50–63, doi:10.1007/BF00571463.

[9] M. E. Conway (1963): *Design of a separable transition-diagram compiler*. Commun. ACM 6(7), pp. 396–408, doi:10.1145/366663.366704.

[10] T. Crolard (1996): *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.

[11] T. Crolard (1999): *A confluent lambda-calculus with a catch/throw mechanism*. Journal of Functional Programming 9(6), pp. 625–647, doi:10.1017/S0956796899003512.

[12] T. Crolard (2001): *Subtractive Logic*. Theoretical Computer Science 254(1–2), pp. 151–185, doi:10.1016/S0304-3975(99)00124-3.

[13] T. Crolard (2004): *A Formulæ-as-Types Interpretation of Subtractive Logic*. Journal of Logic and Computation 14(4), pp. 529–570, doi:10.1093/logcom/14.4.529.

[14] T. Crolard (2015): *A verified abstract machine for functional coroutines - Coq formalization*. Technical Report, CEDRIC - Conservatoire National des Arts et Métiers. Available at `http://cedric.cnam.fr/cpr/crolard/publications`.

[15] P.-L. Curien & H. Herbelin (2000): *The duality of computation*. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM Press, New York, USA, pp. 233–243, doi:10.1145/351240.351262.

[16] O. J. Dahl, E. W. Dijkstra & C. A. R. Hoare (1972): *Structured programming*. Academic Press.

[17] O.-J. Dahl & K. Nygaard (1966): *SIMULA: an ALGOL-based simulation language*. Commun. ACM 9(9), pp. 671–678, doi:10.1145/365813.365819.

[18] O. Danvy (2008): *Defunctionalized Interpreters for Programming Languages*. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'08), ACM Press, New York, USA, pp. 131–142, doi:10.1145/1411204.1411206.

[19] O. Danvy, editor (2007): *Special Issue on the Krivine Machine*. Higher-Order and Symbolic Computation 20(3), doi:10.1007/s10990-007-9021-1.

[20] R. K. Dybvig & R. Hieb (1989): *Engines From Continuations*. Comput. Lang 14(2), pp. 109–123, doi:10.1016/0096-0551(89)90018-0.

[21] H. Eades, A. Stump & R. McCleeary (2016): *Dualized simple type theory*. Submitted to Logical Methods in Computer Science.

[22] S. Fortune, D. Leivant & M. O'Donnell (1983): *The Expressiveness of Simple and Second-Order Type Structures*. J. ACM 30(1), pp. 151–185, doi:10.1145/322358.322370.

[23] D. P. Friedman, C. T. Haynes & M. Wand (1986): *Obtaining Coroutines with Continuations*. Journal of Computer Languages 11(3/4), pp. 143–153, doi:10.1016/0096-0551(86)90007-X.

[24] R. Goré & L. Postniece (2010): *Combining derivations and refutations for cut-free completeness in bi-intuitionistic logic*. Journal of Logic and Computation, doi:10.1093/logcom/exn067.

[25] S. Görnemann (1971): *A logic stronger than intuitionism*. The Journal of Symbolic Logic 36, pp. 249–261, doi:10.2307/2270260.

[26] P. Greussay (1976): *An Iterative Lisp Solution to the Samefringe Problem*. SIGART Bull. 59, pp. 14–14, doi:10.1145/1045270.1045273.

[27] T. G. Griffin (1990): *A formulæ-as-types notion of control*. In: Conference Record of the 17th Annual ACM Symposium on Principles of Programming Langages, pp. 47–58, doi:10.1145/96709.96714.

[28] P. de Groote (1998): *An environment machine for the lambda-mu-calculus*. Mathematical Structure in Computer Science 8, pp. 637–669, doi:10.1017/S0960129598002667.

[29] P. de Groote (2001): *Strong Normalization of Classical Natural Deduction with Disjunction*. In S. Abramsky, editor: *Typed Lambda Calculi and Applications*, *LNCS* 2044, Springer, pp. 182–196, doi:10.1007/3-540-45413-6_17.

[30] A. Grzegorczyk (1964): *A philosophically plausible formal interpretation of intuitionistic logic*. *Nederl. Akad. Wet., Proc., Ser. A* 67, pp. 596–601, doi:10.2307/2271877.

[31] R. Harper, B. F. Duba & D. MacQueen (1993): *Typing first-class continuations in ML*. *Journal of Functional Programming* 3(4), pp. 465–484, doi:10.1017/S095679680000085X.

[32] R. Kashima (1991): *Cut-Elimination for the intermediate logic CD*. Research Report on Information Sciences C100, Institute of Technology, Tokyo.

[33] D. Kimura & M. Tatsuta (2009): *Dual Calculus with Inductive and Coinductive Types*. In Ralf Treinen, editor: *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil*, *LNCS* 5595, Springer, pp. 224–238, doi:10.1007/978-3-642-02348-4_16.

[34] D. E. Knuth (1997): *The Art of Computer Programming, Volume I: Fundamental Algorithms*, 3rd edition edition. Addison-Wesley.

[35] J.-L. Krivine (1994): *Classical logic, storage operators and second order λ-calculus*. *Ann. of Pure and Appl. Logic* 68, pp. 53–78, doi:10.1016/0168-0072(94)90047-7.

[36] D. Leivant (2002): *Intrinsic reasoning about functional programs I: first order theories*. *Annals of Pure and Applied Logic* 114(1-3), pp. 117–153, doi:10.1016/S0168-0072(01)00078-1.

[37] E. G. K. Lopez-Escobar (1983): *A Second Paper "On the Interpolation Theorem for the Logic of Constant Domains"*. *The Journal of Symbolic Logic* 48(3), pp. 595–599, doi:10.2307/2273451. Available at `http://www.jstor.org/stable/2273451`.

[38] C. D. Marlin (1980): *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, doi:10.1007/3-540-10256-6.

[39] J. McCarthy (1977): *Another SAMEFRINGE*. *SIGART Bull.* 61, pp. 4–4.

[40] G. Mints, G. Olkhovikov & A. Urquhart (2013): *Failure of Interpolation in Constant Domain Intuitionistic Logic*. *The Journal of Symbolic Logic* 78, pp. 937–950, doi:10.2178/jsl.7803120. Available at `http://journals.cambridge.org/article_S0022481200126672`.

[41] A. L. de Moura & R. Ierusalimschy (2004): *Revisiting Coroutines*. MCC 15/04, PUC-Rio, Rio de Janeiro, RJ, doi:10.1145/1462166.1462167.

[42] A. L. de Moura, N. Rodriguez & R. Ierusalimschy (2004): *Coroutines in Lua*. *Journal of Universal Computer Science* 10(7), pp. 910–925, doi:10.3217/jucs-010-07-0910.

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

[44] M. Parigot (1993): *Strong normalization for second order classical natural deduction*. In: *Proceedings of the eighth annual IEEE symposium on logic in computer science*, pp. 39–46, doi:10.1109/LICS.1993.287602.

[45] L. Pinto & T. Uustalu (2009): *Proof Search and Counter-Model Construction for Bi-intuitionistic Propositional Logic with Labelled Sequents*. *Automated Reasoning with Analytic Tableaux and Related Methods*, pp. 295–309, doi:10.1007/978-3-642-02716-1_22.

[46] L. Pinto & T. Uustalu (2010): *Relating Sequent Calculi for Bi-intuitionistic Propositional Logic*. In S. van Bakel, S. Berardi & U. Berger, editors: *Proc. of the 3rd Workshop on Classical logic and Computation*, Masarykova Univ., pp. 68–85, doi:10.4204/EPTCS.47.7.

[47] C. J. Prenner (1971): *The Control Structure Facilities of ECL*. *SIGPLAN Not.* 6(12), pp. 104–112, doi:10.1145/800006.807990.

[48] D. Pym, E. Ritter & L. Wallen (2000): *On the intuitionistic force of classical search*. *Theoretical Computer Science* 232(1-2), pp. 299–333, doi:10.1016/S0304-3975(99)00178-4.

[49] C. Rauszer (1974): *Semi-Boolean algebras and their applications to intuitionistic logic with dual operations*. In: *Fundamenta Mathematicae*, 83, pp. 219–249.

[50] C. Rauszer (1980): *An algebraic and Kripke-style approach to a certain extension of intuitionistic logic*. In: *Dissertationes Mathematicae*, 167, Institut Mathématique de l'Académie Polonaise des Sciences, pp. 1–67.

[51] N. J. Rehof & M. H. Sørensen (1994): *The λ_Δ-calculus*. In: *Theoretical Aspects of Computer Software*, *LNCS* 542, Springer-Verlag, pp. 516–542, doi:10.1007/3-540-57887-0_113.

[52] J. H. Reppy (1995): *First-class Synchronous Operations*. In: *Proceedings of the International Workshop on Theory and Practice of Parallel Programming*, *LNCS* 907, Springer-Verlag, London, UK, pp. 235–252, doi:10.1007/BFb0026573.

[53] T. Shimura & R. Kashima (1994): *Cut-Elimination Theorem for the Logic of Constant Domains*. *Math. Log. Q* 40, pp. 153–172, doi:10.1002/malq.19940400203.

[54] C. Strachey & C. P. Wadsworth (1974): *Continuations: A Mathematical Semantics for Handling Full Jumps*. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [58].

[55] T. Streicher & B. Reus (1998): *Classical Logic, Continuation Semantics and Abstract Machines*. *Journal of Functional Programming* 8(6), pp. 543–572, doi:10.1017/S0956796898003141.

[56] The Open Group (1997): *The Single UNIX Specification, Version 2*. Available at `http://www.UNIX-systems.org/online.html`.

[57] A. S. Troelstra (1973): *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. *Lecture Notes in Mathematics* 344, Springer-Verlag, Berlin, doi:10.1007/BFb0066742.

[58] C. P. Wadsworth (2000): *Continuations revisited*. *Higher-Order and Symbolic Computation* 13(1/2), pp. 131–133, doi:10.1023/A:1010074329461.

[59] N. Wirth & J. Mincer-Daszkiewicz (1980): *Modula-2*. ETH Zurich, Schweiz, doi:10.3929/ethz-a-000189918.