A theory of reflexive computation based on soft intuitionistic logic

Hubert Godfroy, Jean-Yves Marion

A theory of reflexive computation based on soft intuitionistic logic

Hubert Godfroy and Jean-Yves Marion

Université de Lorraine, CNRS, LORIA, UMR 7503, Vandoeuvre-lés-Nancy, F-54506, France
Hubert.Godfroy@loria.fr and Jean-Yves.Marion@loria.fr

Abstract. Computational Reflection is a paradigm in which the computational mechanisms controls the different levels of data interpretation. It can be decomposed into two processes: (i) reification, which turns a program into a data and (ii) reflection, which, inversely, installs a data as a program or a context in order to be run. We present a logical account of reflection based on a confluent lambda-calculus. We define two type assignment systems called Soft Intuitionistic Logic (SIL) and Intuitionistic Logic with Promotion (PIL) in which we introduce the exponential modality ! of linear logic. The duality between reification and reflection is to be found in the soft-promotion rule in SIL (resp. in the promotion rule in PIL) and in the dereliction rule. We also show that SIL (or PIL) is a framework in which partial evaluation can be efficiently performed. Finally, we add a control operator, which reifies its evaluation context. Thus a program may have full control of its computational state. We extend the type system with classical logic and semantics is provided by Krivine abstract machine. We establish the soundness of our type system.

1 Introduction

Nowadays, it is common that programs are protected against reverse engineering. One of predominant software protections is the use of packers. Viruses, and more generally malware, incorporate such protections. The run of a packer may be seen as a sequence of code waves. The first wave contains the initial code, which is going to create a second code wave by decrypting or decompressing a data. Then the second code wave in turn will create a third wave, ... and it is not uncommon to see packers that deploy more than 100 waves. A packer is a typical example of a self-modifying program. Another example of a self-modifying program is a Just-in-Time Compiler. A self-modifying program is a program that can modify its own code and that can generate on the fly new codes. Self-modifying programs are inherent to computability. This notion goes back to the second recursion Theorem of Kleene [8, 15]. The article [12] explains the relationship between self-modifying programs and Kleene's amazing construction.

This notion of self-modification is closely related to computational reflection. The intuition behind is that code may be transformed into data, which is

named *reification*, and inversely a data can be transformed into a code, which is named *reflection*. Computational reflection is an important tool in programming language theory. It is the foundation of meta-programming and of Futamura projections [7] (in which an interpreter is specialized to define a compiler). We will show how our framework can be applied to partial evaluation.

Our motivation is that we seek a paradigmatic reflexive type language based on lambda-calculus. As self-modifying programs become more widely used, it is increasingly important to have a better understanding of the notions that underpinned them. We begin with the definition of a confluent lambda calculus Λ_R in which reification and reflection are incorporated. Then, we present a type system SIL, which is sound with respect to Λ_R semantics. The type system SIL is based on intuitionistic logic with the exponential modality !. Implicit computational complexity, and in particular the work of Lafont [11] on Soft Linear Logic, has provided insight on exponential rules. The type system SIL uses the soft promotion rule to build a quoted term, that is to reify an expression. It follows that the dereliction rule amounts to a reflection principle, that is it allows to run a quoted term. Then, we will also see that the soft promotion rule can be replaced by the promotion rule. It turns out that the type system, named PIL, with promotion is equivalent to the system based on modal logic of Davis and Pfenning designed to deal with staged computation [3]. We will also see that we can define a fixpoint that captures the essence of Kleene second recursion theorem and thus define a quine.

Finally, the last section explore the ability of a reflexive program to have access to the computational states that is to have access to both evaluation contexts and current expressions, that are represented as data. To this end, we define a typed lambda calculus SIL* with a control operator (call/cc), which gives access to the reification of the current continuation. The operational semantics is conveyed by the Krivine abstract machine[10]. The type system is built on the algorithmic understanding of classical logic by Griffin [5] and Krivine [9].

The design of typed reflexive λ-calculi is a quite unexplored research line, and so exploring different approaches is interesting. There are two other logical frameworks developed to study staged computation. Taha and his co-authors use Temporal logic [2] with 'next t" and "prev t" markers and type constructor $\bigcirc A$ statement is true for the next stage. A Caml implementation [20] is available. Davis, Pfenning and their co-authors in a series of papers, see [3] and [16], use modal logic to syntactically describe *frozen* terms, that is, closed values which remains to be evaluated. In this framework data are terms of the form $\mathrm{box}(t) : \Box A$ and are in normal form. As we have already said, our study is closed of their works and have been a source of inspiration.

2 Reflection in lambda-calculus

We now present the paradigmatic reflexive language Λ_R. In order to modeling reflection in lambda-calculus, we add two operations that we call *reification* and *reflection*. Reification converts a term t into a quoted term $\langle t \rangle$. The intuition is

that quoted terms represent data and so they cannot be reduced. A quoted term $\langle t \rangle$ may be the abstract syntax tree (AST) of t overs the domain of lists or a string contains an encryption of the AST of t. In this case, the intuition is that run will decrypt $\langle t \rangle$ before running it. The process of converting a reified value $\langle t \rangle$ into a term, which may be then executed, is called reflection. The reflection operator is provided by the construction $\mathsf{let}\langle x \rangle = t' \ \mathsf{in}\ t$. The intention is that $\mathsf{let}\langle x \rangle = \langle s \rangle \ \mathsf{in}\ t$ reduces to $t[x \leftarrow s]$. Thus, the term run defined below executes a reified term by unquoting a quoted term and $\mathsf{run}\langle s \rangle$ evaluates to s

$$\mathsf{run} \stackrel{\text{def}}{=} \lambda y.\, \mathsf{let}\langle x \rangle = y \ \mathsf{in}\ x$$

The set of expressions of Λ_R is defined by the following grammar:

$$\Lambda_R \ni t \stackrel{\text{def}}{=} x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid \mathsf{let}\langle x \rangle = t \ \mathsf{in}\ t$$

All along, we will say that quoted terms are expressions of the form $\langle t \rangle$. In contrast, terms will denote expressions which are not quoted. Expressions refer collectively to both terms and unquoted terms.

That said, if we are not careful, the system may not be confluent. Indeed, consider the expression $(\lambda x.\langle x \rangle)((\lambda z.a)\lambda y.y)$. It could be reduced to either $\langle a \rangle$ or $\langle (\lambda z.a)(\lambda y.y) \rangle$, which are both in normal form. One way to fix the problem is to say that quoted terms are not affected by λ-binders. That is, the variable x in $\lambda x.\langle \lambda y.xy \rangle$ should be free while y is bound. Thereby, the term $(\lambda x.\langle x \rangle)t$ should return $\langle x \rangle$ because the occurrence of x in $\langle x \rangle$ is free inside $\lambda x.\langle x \rangle$. Similarly, the term $\lambda x.\, \mathsf{run}\langle x \rangle$ should reduce to $\lambda y.x$. That is why, the let construction is necessary because it ensures that variables in a quoted term are correctly bounded.

The set of free variables should be carefully defined in order to avoid to capture free variables in quoted terms. Notice that unlike lambda binders, a let bound variable can capture variable in terms as well as in quoted terms. The set of free variables is achieved by defining mutually both the set of free variables in (unquoted) terms (FV^{λ}) and the set of free variables inside quoted terms ($\mathrm{FV}^{\mathsf{let}}$) as follows:

$$\begin{aligned}
\mathrm{FV}^{\lambda}(x) &= \{x\} \\
\mathrm{FV}^{\lambda}(\lambda x.t) &= \mathrm{FV}^{\lambda}(t) \setminus x \\
\mathrm{FV}^{\lambda}(t_1\ t_2) &= \mathrm{FV}^{\lambda}(t_1) \cup \mathrm{FV}^{\lambda}(t_2) \\
\mathrm{FV}^{\lambda}(\langle t \rangle) &= \emptyset \\
\mathrm{FV}^{\lambda}(\mathsf{let}\langle x \rangle = t \ \mathsf{in}\ t') &= \mathrm{FV}^{\lambda}(t) \cup (\mathrm{FV}^{\lambda}(t') \setminus x)
\end{aligned}$$

$$\begin{aligned}
\mathrm{FV}^{\mathsf{let}}(x) &= \emptyset \\
\mathrm{FV}^{\mathsf{let}}(\lambda x.t) &= \mathrm{FV}^{\mathsf{let}}(t) \\
\mathrm{FV}^{\mathsf{let}}(t_1\ t_2) &= \mathrm{FV}^{\mathsf{let}}(t_1) \cup \mathrm{FV}^{\mathsf{let}}(t_2) \\
\mathrm{FV}^{\mathsf{let}}(\langle t \rangle) &= \mathrm{FV}^{\lambda}(t) \cup \mathrm{FV}^{\mathsf{let}}(t) \\
\mathrm{FV}^{\mathsf{let}}(\mathsf{let}\langle x \rangle = t \ \mathsf{in}\ t') &= \mathrm{FV}^{\mathsf{let}}(t) \cup (\mathrm{FV}^{\mathsf{let}}(t') \setminus x)
\end{aligned}$$

As a result, the set of free variables of an term t is $\mathrm{FV}(t) = \mathrm{FV}^{\lambda}(t) \cup \mathrm{FV}^{\mathsf{let}}(t)$. To illustrate this definition, let us consider the term $t \stackrel{\text{def}}{=} (\lambda x.\, \mathsf{run}\, f\langle x\langle \lambda y.y\rangle\rangle)$. We have $\mathrm{FV}(t) = \mathrm{FV}^{\lambda}(t) \cup \mathrm{FV}^{\mathsf{let}}(t) = \{f\} \cup \{x\}$.

From now on, we work as usual with terms *up to renaming of bound variables*, that is up to alpha-conversion. This means that the name of lambda bound variables and the name of let bound variables can be changed to avoid the capture of free variable names wrt FV definition. For example $\mathsf{let}\langle x\rangle = z\ \mathsf{in}\ \lambda x.x\langle x\rangle$ is alpha-equivalent to $\mathsf{let}\langle x_1\rangle = z\ \mathsf{in}\ \lambda x_2.x_2\langle x_1\rangle$. The operation of substitution is then defined thus:

$$\begin{aligned}
x[x \leftarrow s] &= s && \\
y[x \leftarrow s] &= y && y \neq x \\
(\lambda y.t)[x \leftarrow s] &= \lambda y.t[x \leftarrow s] && y \neq x \text{ and } y \notin \mathrm{FV}(s) \\
(tt')[x \leftarrow s] &= t[x \leftarrow s]t'[x \leftarrow s] && \\
(\mathsf{let}\langle y\rangle = t\ \mathsf{in}\ t')[x \leftarrow s] &= \mathsf{let}\langle y\rangle = t[x \leftarrow s]\ \mathsf{in}\ t'[x \leftarrow s] && y \neq x \text{ and } y \notin \mathrm{FV}(s) \\
\langle t\rangle[x \leftarrow s] &= \langle t[x \leftarrow s]\rangle &&
\end{aligned}$$

Reduction rules are based on the contraction of λ-redexes and let-redexes:

$$(\lambda x.t)t' \stackrel{\lambda}{\to} t[x \leftarrow t']$$

$$\mathsf{let}\langle x\rangle = \langle t'\rangle\ \mathsf{in}\ t \stackrel{\mathsf{let}}{\to} t[x \leftarrow t']$$

The set of evaluation contexts of Λ_R expressions is defined as follows:

$$E \stackrel{\text{def}}{=} [\,] \mid E\ t \mid t\ E \mid \lambda x.E \mid \mathsf{let}\langle x\rangle = E\ \mathsf{in}\ t \mid \mathsf{let}\langle x\rangle = t\ \mathsf{in}\ E$$

and reduction rules are extended under contexts as usual:

$$\frac{t \stackrel{\lambda}{\to} t'}{E[t] \stackrel{\lambda}{\to} E[t']} \qquad \frac{t \stackrel{\mathsf{let}}{\to} t'}{E[t] \stackrel{\mathsf{let}}{\to} E[t']}$$

Finally, we define $\to = \stackrel{\lambda}{\to} \cup \stackrel{\mathsf{let}}{\to}$. The reflexive and transitive closure of $\to$ is denoted $\stackrel{*}{\to}$.

It is important to notice that for any context E, $\langle E\rangle$ is not a context. Therefore, it is not possible to reduce a quoted term. This is what is expected since $\langle t\rangle$ is intuitively a data and not a program.

We end by two examples that illustrates two features of the let construction. As we have already said, the let construction allows to unquote an expression. Let us return to the expression run. It allows to execute a quoted term $\langle t\rangle$:

$$\underline{\mathsf{run}\langle t\rangle} \to \underline{(\lambda x.\, \mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y)\langle t\rangle} \to \underline{\mathsf{let}\langle y\rangle = \langle t\rangle\ \mathsf{in}\ y} \to t$$

In turn, t may be executed. The second example illustrates how a variable, which is bound by let, may be replaced in a quoted term.

$$\begin{aligned}
\mathsf{let}\langle x\rangle = \underline{(\lambda x.x)\langle s\rangle}\ \mathsf{in}\ (\lambda xy.y)\ \langle(\lambda y.y)x\rangle\ \langle x\rangle &\to \underline{\mathsf{let}\langle x\rangle = \langle s\rangle\ \mathsf{in}\ (\lambda xy.y)\ \langle(\lambda y.y)x\rangle\ \langle x\rangle} \\
&\to \underline{(\lambda xy.y)\ \langle(\lambda y.y)s\rangle\ \langle s\rangle} \\
&\to \langle s\rangle
\end{aligned}$$

The computation ends because $\langle s\rangle$ is a quoted term. Notice that the reduction of the top redex (with let-binder) is blocked until $(\lambda x.x)\langle s\rangle$ is reduced to a quoted term.

2.1 Confluence

We now state the fact that the system $(\Lambda_R, \to)$ is confluent. Intuitively, the key point is that variables in quoted terms are not captured by outside λ-binders. For example, $(\lambda x.\, \mathsf{let}\langle y\rangle = x \;\mathsf{in}\; \langle y\rangle)\ ((\lambda x.\langle a\rangle)\lambda x.x)$ reduces to $\langle a\rangle$ because the let construction forces to evaluate its first argument $(\lambda x.\langle a\rangle)\lambda x.x$ and then performs a let reduction.

Theorem 1. *$(\Lambda_R, \to)$ is confluent. That is, if $t \overset{*}{\to} t'$ and $t \overset{*}{\to} t''$ then there is an term s such that $t' \overset{*}{\to} s$ and $t'' \overset{*}{\to} s$.*

We decompose the system $(\Lambda_R, \to)$ into two systems $(\Lambda_R, \overset{\lambda}{\to})$ and $(\Lambda_R, \overset{\mathsf{let}}{\to})$ in order to show its confluence. We then prove separately that $(\Lambda_R, \overset{\lambda}{\to})$ and $(\Lambda_R, \overset{\mathsf{let}}{\to})$ are confluent and that both systems commute. By Hindley-Rosen Lemma [17, 6], we conclude that $(\Lambda_R, \to) = (\Lambda_R, \overset{\lambda}{\to} \cup \overset{\mathsf{let}}{\to})$ is confluent.

Property 1 *The reduction rule $(\Lambda_R, \overset{\lambda}{\to})$ is confluent.*

Proof. λ-binders cannot bind variables in quoted terms. Then, without reduction of let-binders, boxed terms are unchangeable. So, $(\Lambda_R, \overset{\lambda}{\to})$ is confluent because it is like pure λ-calculus with constants.

Property 2 *If $t \overset{\mathsf{let}}{\to} t'$ and $t \overset{\mathsf{let}}{\to} t''$ then there is an term s such that $t' \overset{\mathsf{let}}{\to} s$ and $t'' \overset{\mathsf{let}}{\to} s$. As a result, $(\Lambda_R, \overset{\mathsf{let}}{\to})$ is confluent*

Proof. Assume that t contains two let-redexes R and S. Suppose that R is $\mathsf{let}\langle x\rangle = \langle s\rangle \;\mathsf{in}\; s'$ and that we reduce R. There is only one residual of S with respect to R. Indeed, assume that S is a sub-term of R. S can not be a sub-term of $\langle s\rangle$. So S is necessarily a sub-term of s' and so S is not duplicated. The other cases are similar. Since there is no duplication of residuals of S with respect to R, we can indifferently reduce R and S, or the inversely reduce S and R.

Property 3 *Reduction rules $\overset{\lambda}{\to}$ and $\overset{\mathsf{let}}{\to}$ commute. That is if $t \overset{\lambda}{\to}^{*} t'$ and $t \overset{\mathsf{let}}{\to}^{*} t''$, there is t' such that $t' \overset{\mathsf{let}}{\to}^{*} s$ and $t'' \overset{\lambda}{\to}^{*} s$.*

Proof. The reduction of a let-redex does not duplicate (unquoted) λ-redexes. Inversely, the reduction of a λ-redex may create several copies of a let-redex. Therefore, if $t \overset{\lambda}{\to} t'$ and $t \overset{\mathsf{let}}{\to} t''$, then it exists s such that $t' \overset{*\,\mathsf{let}}{\to} s$ and $t'' \overset{\lambda}{\to} s$. As a consequence, $\overset{\lambda}{\to}$ and $\overset{\mathsf{let}}{\to}$ commute.

3 Soft Intuitionistic Logic (SIL)

3.1 Typing reflection

We present a deductive system that we call Soft Intuitionistic Logic (SIL), which is a type assignment system for a proper subset of Λ_R terms. The system is constructed from Intuitionistic Logic to which we add the modality ! that amounts to the exponential modality of Linear Logic.

The types are defined as follows where α ranges over base types:

$$\text{Types} \ni A \stackrel{\text{def}}{=} \alpha \mid A \to B \mid !A$$

The rules of SIL are shown in Figure 1 and are given in sequent calculus style. The assumptions have the form $p_1 : A_1, \ldots, p_n : A_n$ where each p_i is a pattern. A pattern is either a variable x or a *quoted variable* of the form $\langle x \rangle$ where x is a variable. Note that there is no nested quoted variables. A context is a set of assumptions where all patterns are different. The domain of Γ is defined by $\mathrm{dom}(\Gamma) = \{x \mid (x : A) \in \Gamma\} \cup \{x \mid (\langle x \rangle : A) \in \Gamma\}$. Contexts range over Γ, Δ.

$$\textsc{Axiom}\ \frac{}{x : A \vdash x : A} \qquad \textsc{Weakening}\ \frac{\Gamma \vdash t : B}{\Gamma, p : A \vdash t : B}$$

$$\textsc{Contraction 1}\ \frac{\Gamma, y : A, z : A \vdash t : C}{\Gamma, x : A \vdash t[y \leftarrow x][z \leftarrow x] : C} \qquad \textsc{Contraction 2}\ \frac{\Gamma, \langle y \rangle :!A, \langle z \rangle :!A \vdash t : C}{\Gamma, \langle x \rangle :!A \vdash t[y \leftarrow x][z \leftarrow x] : C}$$

$$\to\textsc{Left}\ \frac{\Gamma, x : B \vdash t : C \qquad \Delta \vdash t' : A}{\Gamma, \Delta, f : A \to B \vdash t[x \leftarrow f\ t'] : C} \qquad \to\textsc{Right}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\textsc{SoftPromotion}\ \frac{x_1 : A_1, \cdots, x_n : A_n \vdash t : B}{\langle x_1 \rangle :!A_1, \cdots, \langle x_n \rangle :!A_n \vdash \langle t \rangle :!B} \qquad \textsc{Dereliction}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma, \langle x \rangle :!A \vdash t : B}$$

$$\textsc{Cut}\ \frac{\Gamma \vdash t' : A \quad \Delta, x : A \vdash t : B}{\Gamma, \Delta \vdash t[x \leftarrow t'] : B} \qquad \textsc{Let}\ \frac{\Gamma, \langle x \rangle :!A \vdash t : B}{\Gamma, y :!A \vdash \mathsf{let}\langle x \rangle = y\ \mathsf{in}\ t : B}$$

Fig. 1. Type assignment system SIL.

Take the expression run previously defined. We can see that the type of run is $!A \to A$. Likewise, the expression $\lambda f \lambda x.\, \mathsf{let}\langle s \rangle = f\ \mathsf{in}\ \mathsf{let}\langle t \rangle = x\ \mathsf{in}\ \langle st \rangle$,

which takes a quoted function f and a quoted argument x and re-quotes their application, is of type $!(A \to B) \to !A \to !B$.

The soft promotion rule allows to reify a term t. That is, the soft promotion rule constructs a term $\langle t \rangle$ of type $!A$. Simultaneously, the soft-promotion also quotes all variables in the context, and thus all variables occurring in a quoted terms are marked. The soft promotion rule is inspired by Soft Linear Logic of Lafont [11] using proof nets and by both typed systems defined in Baillot & al. [1] and by Gaboardi & al. [4], which characterized exactly Ptime computations. The multiplexing rule of Soft Linear Logic is meaningless in SIL since there is a contraction rule. The elimination rule for the modality ! is the dereliction rule. It corresponds to a reflective process by turning a quoted terms (aka data) into a term (aka programs). Quoted variables are never bound by a λ but by a let construction. That said, the let rule allows on one hand to use a quoted-variable and on the other hand to introduce a new variable y of modal type (i.e. of type $!A$), which may be then bound by a λ. The let rule is like a border, which waits for a quoted term in order to make a substitution.

The type assignment systems for λ-calculus derived from Linear Logic have to be carefully design in order to be coherent. Indeed, a same λ-term may have different proofs. Thus, in the type assignment system STA of [4] based on Soft Liner Logic, the cut rule is restricted to non-modal types. Similarly, the sequent calculus for Linear Logic of Wadler in [21] has a let rule to deal with the fact that the cut rule and the promotion rule introduce incoherence. It is clear that the system SIL inherits of these difficulties and the proposed solutions.

3.2 Correctness of SIL

The following properties show that SIL is correct with respect to (i) the management of free and bounded variables of Λ_R, which guarantees that the calculus is confluent and (ii) that the typing is preserved during computations.

Property 4

1. *If* $\Gamma \vdash t : A$, *then* $\mathrm{FV}(t) \subseteq \mathrm{dom}(\Gamma)$.
2. *If* $\Gamma \vdash t : A$ *and* $x \in \mathrm{FV}^{\mathsf{let}}(t)$ *then there is* B *such that* $\langle x \rangle : !B \in \Gamma$

Proof. By induction on t.

Notice that if $x \in \mathrm{FV}^{\lambda}(t)$, then $x : B$ is not necessarily in Γ because of Dereliction rule: $\langle x \rangle : !B \vdash x : B$. The property of substitution is expressed thus:

Lemma 1 (Substitution).

1. *If* $\Gamma \vdash t' : A$ *and* $\Gamma, x : A \vdash t : B$, *then* $\Gamma \vdash t[x \leftarrow t'] : B$
2. *If* $\Gamma \vdash \langle t' \rangle : !A$ *and* $\Gamma, \langle x \rangle : !A \vdash t : B$ *then* $\Gamma \vdash t[x \leftarrow t'] : B$

Proof. The proofs are in both cases by induction on t.

Finally, we get the main result which is the soundness of the type system.

Theorem 1 (Subject reduction) *If* $\Gamma \vdash t : A$ *and* $t \to t'$ *then* $\Gamma \vdash t' : A$.

Proof. By induction on the reduction context E.

4 Intuitionistic Logic with Promotion (PIL)

4.1 Promotion

We present PIL, for Intuitionistic Logic with Promotion, which is the system SIL in which the Soft Promotion Rule is replaced by the Promotion rule given below:

$$\textsc{Promotion}\quad \frac{\langle x_1\rangle :!A_1, \cdots, \langle x_n\rangle :!A_n \vdash t : B}{\langle x_1\rangle :!A_1, \cdots, \langle x_n\rangle :!A_n \vdash \langle t\rangle :!B}$$

PIL is a proper extension of SIL, that is SIL $\subsetneq$ PIL. Typically, the term $\lambda y : !A\, \mathsf{let}\langle x\rangle = y\ \mathsf{in}\ \langle\langle x\rangle\rangle$ of type $!A \to !!A$ is derivable in PIL but it is not derivable in SIL. This is due to the fact that the digging rule (i.e. $!A \to !!A$) is not derivable in SIL.

In [3], Davis and Pfenning present a nice type system based on intuitionistic modal logic S4 for studying staged computation. They introduce a modal type constructor $\Box$ which plays the same role that !. Their system is based on sequents with two contexts $\Delta; \Gamma \vdash^e t : A$ where Δ is the modal context and Γ is a non-modal context. Actually, the system PIL based on the promotion rule is equivalent to Davis and Pfenning system.

Theorem 2 *$\Delta; \Gamma \vdash^e t : A$ is derivable iff $\widetilde{\Delta}, \Gamma \vdash t : A$ in* PIL*, where $\widetilde{\Delta} = \{\langle x\rangle :!A \mid x : \Box A \in \Delta\}$.*

This result is not surprising since it is known at least from the works of Schellinx [18] and of Martini and Masini [13] that S4 modal logic can be translated into linear logic.

4.2 A reflexive fixpoint

The system PIL allows to define an interesting reflexive fixpoint Z as follows :

$$Z\langle t\rangle \xrightarrow{Z} t\ \langle Z\langle t\rangle\rangle$$

The type of Z is $!(!A \to A) \to A$. Now, we may express the spirit of Kleene's second recursion Theorem. Recall the intuition. A quoted term is a data. So a function takes as input a data and so it should be defined by an expression of type $!A \to A$. Now, each expression p of type $!A \to A$ has a fixpoint $e \stackrel{\text{def}}{=} Z\langle p\rangle$. Indeed, $p\langle e\rangle \sim e$ where $\sim$ means that both expressions has the same normal form or both diverge. We see that each partial function denoted by an expression of type $!A \to A$ has a fixpoint $\langle e\rangle$. It follows that we can define a self-reproducible expression. For this, let p be $\lambda x.\, \mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y$ and let q be the fixpoint of p. We have that the expression q reduces to q, that is q outputs and runs itself.

5 Staging transformation

5.1 A reflexive typed programming language

Let Σ be a vocabulary that comes with a typing assignation σ such that for each operation $f \in \Sigma$, $\sigma(f)$ is the type of f. Thanks to the vocabulary Σ, we may define first order structures like integers, words, lists and operations like conditional, addition, multiplication. We define $\Lambda_R(\Sigma)$ that extends Λ_R by adding (i) a fixpoint fix and (ii) first order facilities (n-ary constants) with respect to a vocabulary Σ.

$$\Lambda_R(\Sigma) \ni t \stackrel{\text{def}}{=} x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid \mathsf{let}\langle x \rangle = t\ \mathsf{in}\ t \mid \mathsf{fix}\, t \mid f(t, \ldots, t) \qquad \text{where } f \in \Sigma$$

The reduction rules of $\Lambda_R(\Sigma)$ are the ones of Λ_R and rules as given below:

$$\mathsf{fix}\, t \stackrel{\mathsf{fix}}{\to} t\ (\mathsf{fix}\, t) \qquad f(t_1, \ldots, t_n) \stackrel{f}{\to} t' \quad \text{where } f \in \Sigma$$

5.2 Representing expressions

Quoted terms are data with a syntax at the same (meta)-level than Λ_R expressions. Presently, quoted terms don't have any concrete representation and so it is not possible to perform computation on them and for example to define staging transformation, that is to specialize an expression to a data. Therefore, we now give informally an expression-as-data representation of $\Lambda_R(\Sigma)$ expressions. For this, assume given a data domain $\mathcal{D}$ generated by Σ and also assume a coding $\underline{t}$ of an expression t, which is a bijective function from $\Lambda_R(\Sigma)$ to $\mathcal{D}$. Then, $\underline{t}$ is the concrete representation of a quoted term $\langle t \rangle$. An historical example of such a coding is the Gödel numbering but we may also define an encoding function on a more expressive structure using lists as it is explained in Jones book [7].

From now on, we may have an operation $\mathbf{S}$ in Σ that transforms quoted terms, with the proviso that types of $\langle p \rangle$ and of $\mathbf{S}(\langle p \rangle)$ are the same. We illustrate this idea in Figure 2 by defining a term transformation $\mathbf{S}$ that reduces a quoted term like a staging transformation. That is, redexes are reduced when it is possible. As a result, a quoted term like $\langle \lambda f(\lambda x.(f\,x))a \rangle$ is reduced to $\langle \lambda f(f\,a) \rangle$.

5.3 An example of staging transformation

An important benefit of having a program transformation defined over the encoding of quoted terms is that partial evaluation can be performed efficiently. Take as an example the power function $\mathsf{pow}\, n\ a$ such $\mathsf{pow}\, n\ a = a^n$. If we know that the exponent, say 2, we would like to *fully specialized* the function pow and obtain $\langle \lambda a.a \underline{\times} a \rangle$. For this just take the following definition of pow:

$$\begin{array}{l}
\mathsf{pow} = \mathsf{fix}\, \lambda p.\lambda n. \\
\quad \mathsf{if}\ (n = 0) \\
\quad\ \mathsf{then}\ \langle \lambda a.1 \rangle \\
\quad\ \mathsf{else}\ \mathsf{let}\langle x \rangle = p\ (n-1)\ \mathsf{in}\ \mathbf{S}(\langle \lambda a.a \underline{\times} (x\ a) \rangle)
\end{array}$$

$$\frac{\mathbf{S}(\underline{t}) \overset{\mathbf{S}}{\to} \underline{t'}}{\mathbf{S}(\underline{\lambda x.t}) \overset{\mathbf{S}}{\to} \underline{\lambda x.t'}}$$

$$\frac{\mathbf{S}(\underline{t_i}) \overset{\mathbf{S}}{\to} \underline{t'_i} \qquad f(t'_1, \cdots, t'_n) \overset{f}{\to} t \qquad \mathbf{S}(t) \overset{\mathbf{S}}{\to} t'}{\mathbf{S}(\underline{f(t_1, \cdots, t_n)}) \overset{\mathbf{S}}{\to} \underline{t'}}$$

$$\frac{\mathbf{S}(\underline{t_i}) \overset{\mathbf{S}}{\to} \underline{t'_i} \qquad t'_1 = \lambda x.t''_1 \qquad t'_1\ t'_2 \overset{\lambda}{\to} t''_1[x \leftarrow t'_2] \qquad \mathbf{S}(\underline{t''_1[x \leftarrow t'_2]}) \overset{\mathbf{S}}{\to} \underline{t'}}{\mathbf{S}(\underline{t_1\ t_2}) \overset{\mathbf{S}}{\to} \underline{t'}}$$

$$\frac{\mathbf{S}(\underline{t_i}) \overset{\mathbf{S}}{\to} \underline{t'_i} \qquad t'_1 = \langle t''_1 \rangle \qquad \mathsf{let}\langle x \rangle = t'_1\ \mathsf{in}\ t'_2 \overset{\mathsf{let}}{\to} t'_2[x \leftarrow t''_1] \qquad \mathbf{S}(\underline{t'_2[x \leftarrow t''_1]}) \overset{\mathbf{S}}{\to} \underline{t'}}{\mathbf{S}(\underline{\mathsf{let}\langle x \rangle = t_1\ \mathsf{in}\ t_2}) \overset{\mathbf{S}}{\to} \underline{t'}}$$

$$\textit{Otherwise :} \qquad \mathbf{S}(\underline{t}) \overset{\mathbf{S}}{\to} \underline{t}$$

Fig. 2. The (meta)-rules of the program transformation **S**

where **S** is the program transformation defined previously. The function pow has the type int $\to$!(int $\to$ int) in SIL. By a straightforward induction on n, we can prove that pow n is fully specialized. So for example, pow 3 is specialized to $\langle \lambda a.a \underline{\times}\ a \underline{\times}\ a \rangle$.

6 A typed reflexive theory of control

Until now, we have focused on the process of reflecting a quoted term or on reifying a term. The next step is to a give to a program the ability to reflect and reify its own context. This is an important feature in the design of a programming language by "bootstrapping". More generally, the design of self-modifying programs necessitates to control contexts of execution. A typical example which comes from the malware programming world is a payload (i.e. the code of the virus) which is run piece by piece inside a sequence of waves of interpreters. Each wave may just run another layer of interpretation and may call a part of the code of the payload and returns a value to the context of the current wave or to the context of a previous wave.

6.1 Syntax and semantics

We define Λ^*_R which extends Λ_R by adding continuations as follows

$$\Lambda^*_R \ni t \overset{\text{def}}{=} x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid \mathsf{let}\langle x \rangle = t\ \mathsf{in}\ t \mid k_\pi \mid \mathcal{E}$$

Stacks (or evaluation contexts) are defined simultaneously thus

$$\pi = [] \mid t.\pi \mid (x, t).\pi$$

A stack is a finite sequence $t_1 \cdots t_n.[]$ of closed expressions or of couples (x,t) where $\mathrm{FV}(t) \subset \{x\}$, that ends with the bottom of the stack $[]$. Expressions k_π are *continuations*. They are constants indexed by stacks.

Semantics of Λ_R^* is given by a Krivine's machine [10] that performs a Call-By-Name (CBN) computation:

$$\begin{array}{llllll}
t\ t' & \star & \pi \to t & \star & t'.\pi \\
\lambda x.t & \star & t'.\pi \to t[x \leftarrow t'] & \star & \pi \\
\mathsf{let}\langle x\rangle = t\ \mathsf{in}\ t' & \star & \pi \to t & \star & (x,t').\pi \\
\langle t\rangle & \star & (x,t').\pi \to t'[x \leftarrow t] & \star & \pi \\
\mathcal{E} & \star & t.\pi \to t & \star & \langle k_\pi\rangle.[] \\
k_{\pi'} & \star & t.\pi \to t & \star & \pi'
\end{array}$$

The main difference with a standard Krivine's abstract machine is the control operator $\mathcal{E}$. When $\mathcal{E}$ is run[1], the current context is reified and thus $\langle k_\pi\rangle$ is pushed on the current stack. Thus, a program may eventually read or modify its evaluation context because it is a data. Conversely, in order to run a continuation $k_{\pi'}$, we need first to reflect the quoted stack $\langle k_{\pi'}\rangle$ thanks to the let rule. When a let rule is triggered, the expression t is first evaluated and (x,t') is pushed on the stack. Then, when a quoted term is met $\langle t\rangle$, the variable x is replaced by t in t'.

A state of Krivine abstract machine is a pair $t \star \pi$ where t is a closed expression and π is a stack. The initial state is $t \star []$. There are four kinds of halting states: $\lambda x.t \star []$, $\langle t\rangle \star []$, $\mathcal{E} \star []$ and $k_\pi \star []$, which correspond to the end of a normal computation. All other final states are error states. For example, $\lambda x.t \star (x,t).\pi$ is an error state because (x,t) is not a term.

6.2 A type system for reflexive continuations

Following Krivine's approach [9], we define a type system SIL^* to type a subset of Λ_R^* expressions. SIL^* is an extension of SIL. Types are defined thus.

$$A \stackrel{\text{def}}{=} A \to A | !A \mid \bot$$

The constant type $\bot$ is the "empty type", that is the type of terms which do not return anything to their immediate context. For instance, $k_\pi\ t$ evaluates t in π and is of type $\bot$ because it returns nothing locally.

[1] The control operator is dubbed $\mathcal{E}$ to recall Kleene's enumerator E which satisfies $E(\#M) = M$ where $\#M$ is an encoding of M. See Barendregt's book p.167

Type rules of SIL* are inspired by Selinger [19] and comprise the following rules for constants and stacks in addition of the type rules of SIL

$$\textsc{Call/CC}\ \frac{}{\Gamma \vdash \mathcal{E} : (!(A \to \bot) \to \bot) \to A} \qquad \textsc{Throw}\ \frac{\pi : A \vdash}{\Gamma \vdash k_\pi : A \to \bot}$$

$$\textsc{Bot}\ \frac{}{[] : \bot \vdash} \qquad \textsc{Top}\ \frac{}{[] : A^{\text{top}} \vdash}$$

$$\textsc{TermCons}\ \frac{\vdash t : A \qquad \pi : B \vdash}{t.\pi : A \to B \vdash} \qquad \textsc{LetCons}\ \frac{\langle x \rangle : !A \vdash t : B \qquad \pi : B \vdash}{(x,t).\pi : !A \vdash}$$

The type of $\mathcal{E}$ is a variation of the double negation rule in which the modality ! indicates that it is not the stack which is stored but a data that represents a stack. Here the rule Top where A^{top} is the top-level type of the entire program as in [19]. The type of a stack is the type of a term that can use the stack as an evaluation context.

Finally, a state $t \star \pi$ is well typed if both the expression t and the stack π are of same type:

$$\textsc{Proc}\ \frac{\vdash t : A \qquad \pi : A \vdash}{\vdash t \star \pi}$$

6.3 Soundness of the type system

Now we establish that the evaluation of closed and well-typed expression t in SIL* by the Krivine abstract machine is correct.

Theorem 3 (Type soundness) *If t is closed and is well typed in* SIL**, then there is no sequence of the Krivine's abstract machine from $t \star []$ leading to an error state.*

To prove this theorem, we use a modified version of the subject reduction property which claims that well typing property is preserved all along the computation (Lemma 3). Moreover if t is closed and well typed, then $t \star []$ is well typed. And finally no error state is well typed (Lemma 2).

Lemma 2. *No error state is well typed.*

Proof. Error states are of the forms $\lambda x.t \star (x',t').\pi$, $\langle t \rangle \star t'.\pi$, $\mathcal{E} \star (x,t).\pi$ and $k_\pi \star (x,t').\pi'$. Each error cannot be typed because types $A \to B$ and $!C$ are different.

SIL* enjoys subject reduction.

Lemma 3 (Subject reduction). *If $t \star \pi$ is well typed and $t \star \pi \to t' \star \pi'$ then $t' \star \pi'$ is well typed.*

Proof. Substitution Lemma 1 is still valid. Then the theorem is proved by case analysis on the evaluation rule:

- Suppose that $\mathsf{let}\langle x\rangle = t \ \mathsf{in}\ t' \star \pi$ is well typed. Then $\langle x\rangle :!A \vdash t' : B$ and $\vdash t :!A$ and $\pi : B \vdash$. It follows by rule LetCons that $(x,t').\pi :!A \vdash$. So $t \star (x,t').\pi$ is well typed.
- Suppose that $\langle t\rangle \star (x,t').\pi$ is well typed. Then $\vdash t : A$, $\langle x\rangle :!A \vdash t' : B$ and $\pi : B \vdash$. By substitution lemma, $\vdash t'[x/t] : B$. Then $t'[x/t] \star \pi$ is well typed.
- Suppose that $\mathcal{E} \star t.\pi$ is well typed. Then $t.\pi : (!(A \to \bot) \to \bot) \to A) \vdash$. It follows that $\vdash t :!(A \to \bot) \to \bot$ and $\pi : A \vdash$. Since $\vdash \langle k_\pi\rangle :!(A \to \bot)$, we have $\langle k_\pi\rangle.[] :!(A \to \bot) \to \bot \vdash$. So $t \star \langle k_\pi\rangle.[]$ is well typed.

The other cases are similar.

6.4 Going up and down

The system SIL* provides mechanisms to have full access to the state of the computation, that is the expression under evaluation and its context. This access is provided by reification. Similarly, reflection can specify in which evaluation context an expression is evaluated. As previously said, it is common to specify a self-modifying program as a reflective tower in which the term at wave n spawns a term at wave $n+1$ by executing a quoted term. Then, the expression run at wave $n+1$ may have access to the evaluation context of wave n, and actually it may have access to each evaluation context below it, that is from the wave 1 to the wave $n+1$.

We illustrate this scenario by an example. For this, we reread and adapt Mendhekar and Friedman [14] approach. We first define a reflective operator $\otimes :!(A \to \bot) \to !A \to \bot$ which takes a reified evaluation context and a reified expression and create a new wave by spawning the input expression in the given evaluation context:

$$\otimes \stackrel{\text{def}}{=} \lambda K.\lambda T.\, \mathsf{let}\langle k\rangle = K \ \mathsf{in}\ \mathsf{let}\langle t\rangle = T \ \mathsf{in}\ k\ t.$$

Thus, if $\langle T_{n+1}\rangle$ encodes a given expression T_{n+1} and $\langle k_{\pi_{n+1}}\rangle$ encodes a given evaluation context π_{n+1}, $\otimes\langle k_{\pi_{n+1}}\rangle\langle T_{n+1}\rangle$ generates the wave $n+1$ by spawning the state $(T_{n+1} \star \pi_{n+1})$ as follows:

$$\otimes\langle k_{\pi_{n+1}}\rangle\langle T_{n+1}\rangle \ \star\ \pi \quad \stackrel{*}{\to} \quad k_{\pi_{n+1}} T_{n+1} \ \star\ \pi \quad \stackrel{*}{\to} \quad T_{n+1} \ \star\ \pi_{n+1}$$

The reflective operator $\otimes$ allows to go from wave n to wave $n+1$. Intuitively, we may define the wave 1 as the sequence of states starting from the initial state $(t \star [])$ to the first state $(\otimes\langle k_{\pi_1}\rangle\langle T_1\rangle) \star \pi$ containing $\otimes$. At this moment, the second wave begins until it reaches again a state like $(\otimes\langle k_{\pi_2}\rangle\langle T_2\rangle) \star \pi'$, and so on. The intuition is that the reflective operator $\otimes$ is not executed inside a wave and so no data are run.

The second step of the construction consists in providing access to the evaluation context of the wave n to T_{n+1} at wave $n+1$. For this, we suppose that

T_{n+1} has a free variable k which will be replace by a continuation pointing to the evaluation context of the previous wave. The wave n expression T_n spawns wave $n+1$:

$$T_n \stackrel{\text{def}}{=} \mathcal{E}(\lambda K.\, \mathsf{let}\langle k\rangle = K \;\mathsf{in}\; \otimes \langle k_{\pi_{n+1}}\rangle\ \langle T_{n+1}\rangle).$$

Suppose that the stack π_n is of type A_n. Then, T_n is of type A_n. The run of T_n in its evaluation context π_n proceeds as follows:

1. T_n captures a reified version of its context:

$$\begin{aligned} &\mathcal{E}(\lambda K.\, \mathsf{let}\langle k\rangle = K \;\mathsf{in}\; \otimes \langle k_{\pi_{n+1}}\rangle\ \langle T_{n+1}\rangle)\ \star\ \pi_n \\ \to\ &\lambda K.\, \mathsf{let}\langle k\rangle = K \;\mathsf{in}\; \otimes \langle k_{\pi_{n+1}}\rangle\ \langle T_{n+1}\rangle\ \star\ \langle k_{\pi_n}\rangle.[] \\ \to\ &\mathsf{let}\langle k\rangle = \langle k_{\pi_n}\rangle \;\mathsf{in}\; \otimes \langle k_{\pi_{n+1}}\rangle\ \langle T_{n+1}\rangle\ \star\ [] \end{aligned}$$

2. T_n sends its reified context to data $\langle T_{n+1}\rangle$ by binding k.

$$\otimes\langle k_{\pi_{n+1}}\rangle\ \langle T_{n+1}[k \leftarrow k_{\pi_n}]\rangle\ \star\ [] \quad \stackrel{*}{\to} \quad T_{n+1}[k \leftarrow k_{\pi_n}]\ \star\ \pi_{n+1}$$

References

1. Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *In Proc. FoSSaCS, Springer LNCS 2987*, pages 27–41. Springer, 2004.
2. Rowan Davies. A temporal-logic approach to binding-time analysis. *Logic in Computer Science*, pages 184–195, Jul 1996.
3. Rowan Davies and Frank Pfenning. A Modal Analysis of Staged Computation. *Principles of programming languages*, 1996.
4. Marco Gaboardi, Simona Ronchi, and Della Rocca. A soft type assignment system for λ-calculus, 2007.
5. Timothy G. Griffin. A Formulae-as-type Notion of Control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.
6. J. Roger Hindley. *The Church-Rosser Property and a Result in Combinatory Logic.* PhD thesis, University of Newcastle, 1964.
7. Neil D. Jones. *Computability and Complexity.* MIT Press, 1997.
8. S. C. Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):150–155, 1938.
9. Jean-Louis Krivine. Dependent choice, quote and the clock. *Theoretical Computer Science*, 2003.
10. Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007.
11. Yves Lafont. Soft Linear Logic and Polynomial Time. *Theoretical Computer Science*, 318:2004, 2002.
12. Jean-Yves Marion. From Turing machines to computer viruses. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 370(1971):3319–3339, 2012.
13. Simone Martini and Andrea Masini. A Modal View of Linear Logic. *The Journal of Symbolic Logic*, 59(3):pp. 888–899, 1994.

14. Anurag Mendhekar and Dan Friedman. Towards a theory of reflexive programming languages. *Worskshop on Reflection and Meta-level Architectures*, 1993.
15. Yiannis N. Moschovakis. Kleene's amazing second recursion theorem. *Bulletin of Symbolic Logic*, 16(2):189–239, 2010.
16. Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *Transactions on Computational Logic*, February 2007.
17. Barry K. Rosen. Tree-Manipulating Systems and Church-Rosser Theorems. *J. ACM*, 20(1):160–187, January 1973.
18. Harold Schellinx. *A Linear Approach to Modal Proof Theory*, pages 33–43. Springer Netherlands, Dordrecht, 1996.
19. Peter Selinger. From Continuation Passing Style to Krivine Abstract Machine. May 2003.
20. Walid Taha. A Gentle Introduction to Multi-stage Programming. *Domain-Specific Program Generation*, 2004.
21. Philip Wadler. A syntax for linear logic. *International Conference on the Mathematical Foundations of Programming Semantics*, April 1993.