Appeared in Proc.of FPCA'91, LNCS Vol.523, pp.241-258.

Syntactic Detection of Single-Threading using Continuations*

Pascal Fradet
IRISA / INRIA
Campus de Beaulieu, 35042 Rennes Cedex, France
fradet@irisa.fr

Abstract

We tackle the problem of detecting global variables in functional programs. We present syntactic criteria for single-threading which improves upon previous solutions (both syntactic and semantics-based) in that it applies to higher-order languages and to most sequential evaluation strategies. The main idea of our approach lies in the use of continuations. One advantage of continuation expressions is that evaluation ordering is made explicit in the syntax of expressions. So, syntactic detection of single-threading is simpler and more powerful on continuation expressions. We present the application of the analysis to the compilation of functional languages, semantics-directed compiler generation and globalization-directed transformations (i.e. transforming non-single-threaded expressions into single-threaded ones). Our results can also be turned to account to get single-threading criteria on regular λ-expressions for different sequential evaluation orders.

1 Introduction

Single-threading is a property allowing function parameters or semantics domains to be implemented by a global variable. This optimization can have drastic effects on the efficiency of both functional language and semantics-directed compilers. In particular, single-threading can be exploited for register allocation and for the efficient implementation of contiguous data structures.

Imperative language compilers make great use of registers [1], whereas functional language compilers do not take much advantage from register allocation because of frequent context switching. A function which is single-threaded in its argument (we will say that the argument is globalizable) can be compiled using a global register to hold its argument value. A recursive call with a new argument results in a register updating and registers are used throughout the reduction of recursive functions without having to be saved in the stack.

Contiguous data structures, such as arrays, are useful because they can be accessed in constant time. A functional implementation of arrays involves making a new copy before each update (since the original array can be referenced later on). This can clearly be a source of extreme inefficiency. A program using an array in a single-threaded fashion can be implemented using destructive updates on a global array variable. This optimization has a counterpart in semantics-directed compiler generation [11]: the store appears as an argument of the semantic function and a naïve implementation involves a duplication of the store each time the update function is called. In order to derive realistic compilers from denotational specifications one should detect that the store can safely be modified in place. This can be done by checking that the semantic definition is single-threaded in its store domain.

Two approaches have been considered for the detection of single-threading: syntactic analysis [19] and semantic analysis (or abstract interpretation) [2,3,9,18]. In both cases, analyses are approxi-

* Part of this research was done during a visit to Kansas State University (thanks to David Schmidt).

mate since single-threading is not a decidable property. Abstract interpretation often produces more precise results at the cost of an exponential worst case complexity. For this particular problem, it is not clear whether the gain is worth the cost. Syntactic analysis turns out to be sufficient in most common cases and we choose it here for two reasons:

- it is typically a linear time analysis,
- in case of failure, it indicates the faulty subexpression and detects (for free) subexpressions satisfying the property. One can take advantage of this information to perform program transformations (section 4.3) or local optimizations (section 4.1).

The main idea of our approach lies in the use of continuations. Single-threading depends heavily on the evaluation strategy; for example a single-threaded expression using left-to-right call-by-value may turn out to be non single-threaded using right-to-left call-by-value. One advantage of continuation expressions is that evaluation ordering is made explicit in the syntax of expressions. Therefore, syntactic detection of properties depending on the computation rule is likely to be simpler and more powerful on this kind of expressions. We should point out that we do not consider here continuations as a programming tool (in particular, we are not interested in first-class continuations) but as a way of formalizing part of the implementation process, namely the evaluation strategy. Our analysis applies to continuation expressions that are supposed to be produced by a continuation passing style (CPS) transformation. However, our results can be turned to account to get single-threading criteria on regular λ-expressions for different evaluation orders. This is one important advantage of our approach: it can be applied to most sequential evaluation strategies.

Section 2 introduces the syntax of continuation expressions and studies their reduction. In section 3, we present our criteria for single-threading and the associated global variable transformation. Section 4 is devoted to the application of the analysis to functional language implementation, semantics-based compilation and goal-directed transformation. Section 5 describes the generation of sufficient conditions for single-threading on ordinary λ-expressions according to several evaluation orders. We conclude with an overview of related works and a discussion on possible extensions.

2 Continuation expressions

Continuations have been primary used in denotational semantics to model unrestricted jumps [21,22]. Continuation semantics take an additional argument (a continuation) representing control, that is, the evaluation ordering of program constructs. The same concept is used to compile the computation rule of functional programs [4,5,14,16]. CPS transformations produce continuation expressions reducible without dynamic search for the next redex. Typically, a CPS compiler transforms an expression E into an expression E_c taking a continuation as argument and applying it to the result of evaluating E. For example, each operator *opm* such that $opm\ V_1 \ldots V_m \rightarrow_\delta N$, is transformed into a new operator *opmc* such that $opmc\ C\ V_1 \ldots V_m \rightarrow_\delta C\ N$.

The language used in this paper is described in Figure 1. It is general enough to support the compilation of most sequential evaluation orders and standard optimizations of CPS transformations.

Expressions **Types**

$V^\nu ::= K^\beta \mid v^\nu \mid opmc^{(\nu \rightarrow \kappa) \rightarrow \nu 1 \ldots \rightarrow \nu m \rightarrow \kappa} \mid \lambda c^{\kappa 1}.C^{\kappa 2} \mid rec\ f^\nu = V^\nu$ $\quad \nu ::= \beta \mid \kappa_1 \rightarrow \kappa_2 ; \quad \beta ::= bool \mid int \mid [int] \mid \ldots$

$C^\kappa ::= fin^{\nu \rightarrow \mathcal{A}} \mid c^\kappa \mid (ifc\ C_1^\kappa\ C_2^\kappa)^{\ bool \rightarrow \kappa} \mid \lambda v^\nu.C^\kappa \mid C^{\nu \rightarrow \kappa}\ V^\nu \mid V^{\kappa 1 \rightarrow \kappa 2}\ C^{\kappa 1}$ $\quad \kappa ::= \mathcal{A} \mid \nu \rightarrow \kappa$

Figure 1 Syntax of Continuation Expressions

Type β stands for first-order types, ν for value types, κ for continuation types and $\mathcal{A}$ for answers. An expression is either a value V^{ν} or a continuation C^{κ}. A value is either a first-order constant *K*, a variable *v*, a strict operator of arity m *opmc*, a functional value $\lambda c^{\kappa 1}.C^{\kappa 2}$ or a recursive definition (note that values are weak normal forms). A continuation is either the final continuation $fin^{\nu \to \mathcal{A}}$, a continuation variable with the unique name *c*, a conditional *ifc* taking two continuation expressions and a boolean, a functional continuation $\lambda v^{\nu}.C^{\kappa}$ or an application ($V^{\kappa 1 \to \kappa 2}\ C^{\kappa 1}$ or $C^{\nu \to \kappa}\ V^{\nu}$). In the remainder of the paper we often omit types to make reading easier.This syntax generalizes usual continuation expressions in two ways:

- One standard CPS optimization, which is very important in practice, is to avoid to introduce a continuation for each λ-abstraction. For example, if an expression $\lambda v_1 ... \lambda v_n.E$ is known to be fully applied to *n* arguments, then a single continuation is sufficient and the transformed expression is of the form $\lambda c.\lambda v_1 ... \lambda v_n.E_c$. Thus, function values which usually have type $\kappa \to \nu \to \mathcal{A}$ (that is, take a continuation, a value and yields an answer) have here the generalized type $\kappa \to \nu_1 \to \ldots \to \nu_n \to \mathcal{A}$. For the same concerns, continuation types, usually of the form $\nu \to \mathcal{A}$, are generalized to $\nu_1 \to \ldots \to \nu_n \to \mathcal{A}$.
- Usually, continuation expressions are of the form $E_1 \ldots E_n$ (E_is being weak normal forms) Here, we allow nested applications in continuations. Expressions can be of the form $V\ C\ V_1 \ldots V_n$, *C* being itself of the form $U\ C'\ U_1 \ldots U_n$.

The factorial function can be written in this continuation passing style language as follows:

rec factc = λc.λn. eqc (ifc (c 1) (subc (factc (multc c n)) n 1)) n 0

The reduction rules for this language are the following:

(β) $(\lambda x.E)F \to_{\beta} E[F/x]$

(fix) $rec\ f = V \to_{fix} V[rec\ f = V/f]$

(δ) $opmc\ C\ V_1 \ldots V_m \to_{\delta} C\ N$ $\qquad$ $ifc\ C_1\ C_2\ True \to_{\delta} C_1$ $\qquad$ $ifc\ C_1\ C_2\ False \to_{\delta} C_2$

No reduction rule is specified for the continuation *fin* and expressions *fin V* are considered as normal forms.

The key property (Property 1) of those expressions is that they can be evaluated using a simpler evaluation order than normal order. It is sufficient to reduce at each step the head operator thus avoiding the need of a dynamic search for the next redex. This evaluation scheme, called First, is formally defined (Figure 2) by a set of axioms (no inference rule is needed).

$opmc\ C\ V_1 \ldots V_m\ V_{m+1} \ldots V_n \xrightarrow[F]{1} C\ N\ V_{m+1} \ldots V_n$

$ifc\ C_1\ C_2\ True\ V_1 \ldots V_n \xrightarrow[F]{1} C_1\ V_1 \ldots V_n$

$ifc\ C_1\ C_2\ False\ V_1 \ldots V_n \xrightarrow[F]{1} C_2\ V_1 \ldots V_n$

$(\lambda x.E)\ F\ V_1 \ldots V_n \xrightarrow[F]{1} E[F/x]\ V_1 \ldots V_n$

$(rec\ f = V)\ V_1 \ldots V_n \xrightarrow[F]{1} V[rec\ f = V/f]\ V_1 \ldots V_n$

Figure 2 First evaluation strategy

Property 1 *An expression E has a weak head normal form (whnf) W iff E has a whnf W' (W cnv W') such that* $E \xrightarrow[F]{*} W'$.

Proof: Using the standardization theorem and the fact that any expression of type ν is a wnf. The only kind of expression being a First normal form without being a whnf is a strict operator applied to unevaluated arguments (e.g. *plus 1 (plus 2 3)*). But syntactic restrictions (Figure 1) enforce that strict operators always have their arguments in normal form (e.g. the former expression would be written *plusc (plusc fin 1) 2 3*), so First normal form and whnf are equivalent for our language.

Example: We describe here the first steps of the reduction by First of the factorial function applied to the continuation *fin* and argument 1.

(rec factc = λc.λn. eqc (ifc (c 1) (subc (factc (multc c n)) n 1)) n 0) fin 1

$\xrightarrow[F]{}$ *eqc (ifc (fin 1) (subc ((rec factc = λc.λn...)(multc fin 1)) 1 1)) 1 0*

$\xrightarrow[F]{}$ *ifc (fin 1) (subc ((rec factc = λc.λn...) (multc fin 1)) 1 1)) False*

$\xrightarrow[F]{}$ *subc ((rec factc = λc.λn...) (multc fin 1)) 1 1*

$\xrightarrow[F]{}$ *(rec factc = λc.λn...) (multc fin 1) 0*

… and so on until … $\xrightarrow[F]{}$ *fin 1*

It is often helpful to look at continuation expressions operationally. An expression $V\ C\ V_1 \ldots V_n$ can be seen as a stack machine sate, *V* being the instruction to reduce, the continuation *C* representing the remainder of the program, and $V_1,\ldots,V_n$ the stack. Actually, after a few supplementary transformations (e.g. abstraction using combinators), one can get true generic stack machine code from those expressions [5,6]. The important point for the single-threading detection is that for any application $V^{\kappa 1 \to \kappa 2}\ C^{\kappa 1}$ we know that *V* is evaluated first and then applies continuation *C* to its result. We show in the next section how this information can be exploited.

3 Single-Threading Detection and Globalization Transformation

Let us take an example to introduce the problem and our solution. Let f be a recursive function defined by:

rec f = λa.λi.λp. if (eq i 0) a (f (update a i p) (sub i 1) (mult p (access a i)))

Function *f* can be used to replace each element *a[i]* of the array *a* by the product of its successors; for example *f [4,3,2] 3 1= [6,2,1]*. A naïve implementation of *f* would produce code copying the array before each update. We want to detect whether the first argument can be globalized and updated destructively. First, we should notice that the entity to be represented by a global variable is the sequence of arrays (*a, update a i p, update (update a i p) (i-1) (a[i]*p),…*). We rely on types to characterize single-threading properties. Let $\rho \to \text{int} \to \text{int} \to \rho$ be the type of *f*, therefore each element of the above sequence has type ρ and the problem is to detect whether type ρ can be implemented by a global variable. (Throughout the paper, type ρ denotes the candidate for globalization. We sometimes say that a function is single-threaded in an argument (or parameter) assuming that this argument can be characterized by its type). Assuming a call-by-value (cbv) implementation of function *f*, two cases arise:

- the update is done before the access (left-to-right cbv). In this case, *f* is not single-threaded and we cannot use destructive updates,

- the access is done before the update (right-to-left cbv) and the array can be modified in place.

Such cases are easily detected on CPS versions of *f*. Considering left-to-right call-by-value, *f* is transformed into:

rec f = $\lambda c.\lambda a.\lambda i.\lambda p.$ *eqc (ifc (c a)*

(updatec (subc (accessc (multc ($\lambda k.\lambda l.\lambda m.$*f c m l k) p) a i) i 1) a i p)) i 0*

We will say that an expression *F modifies* ρ if its reduction may involve the creation of a new instance of type ρ. In particular, an operator *opmc* of type $(\rho \to \kappa_1) \to \kappa_2$, which yields an element of type ρ (since its continuation has type $\rho \to \kappa_1$), will be a modifier (of ρ). The criterion used here to detect that *f* is not single-threaded in ρ is that a free variable of type ρ (namely *a*) appears in the continuation of a modifier (*updatec*). In other words, the structure would be modified whereas it is still referenced (by a free variable); therefore it cannot be updated in place.

Using right-to-left call-by-value, *f* becomes:

rec f = $\lambda c.\lambda a.\lambda i.\lambda p.$ *eqc (ifc (c a) (accessc (multc (subc (updatec (f c) a i p) i 1) p) a i)) i 0*

Here, no free variable of type ρ appears in the continuation of the modifier *updatec*, and we will see later that this expression is indeed single-threaded in ρ.

3.1 Single-threading criteria

Our criteria are in the form of a predicate Δ such that $\Delta_\rho(E)$ implies that *E* is single-threaded in type ρ. Before giving a formal and comprehensive account of Δ, let us introduce it in a more intuitive way.

Let $V^{\kappa 1 \to \kappa 2}\ C^{\kappa 1}\ V_1 \ldots V_n$ be the expression to analyze. Operationally, single-threading implies that every element of type ρ in the stack $V_1 \ldots V_n$ can be represented by a single global variable throughout the reduction. First, we must check that all elements of type ρ present in the stack are equal. To this purpose, predicate Δ always keeps the last encountered ρ-value. A new value or a modification will be accepted only when

(i) There is no reference (free variables) to an "old" ρ-value occurring in the continuation. Predicate $nfv_\tau(E)$ indicates that *E* does not contain free variables of type τ (we also use $fv_\tau(E)$ to denote the opposite).

(ii) No ρ-value occurs in the stack. This is checked on the type of $(V\ C)^{\kappa 2}$ by predicate $nt_\rho(\kappa 2)$ defined by $n_\beta(\tau) = \textit{if}\ \tau \equiv \tau_1 \to \tau_2\ \textit{then}\ \tau_1 \not\equiv \beta \wedge nt_\beta(\tau_2)\ \textit{else True}$. Thus $nt_\rho(\kappa 2)$ means that there is no ρ-value among $V_1, \ldots, V_n$.

This explains the three following criteria:

- General expressions of the form $V^{\kappa 1 \to \kappa 2}\ C^{\kappa 1}$ are single-threaded in ρ if *V* and *C* are single-threaded in ρ and if *V* modifies ρ then its continuation *C* does not contain free variables of type ρ (rule (Δ9)). For example, expression $\lambda c.\lambda x^\rho.\ succ^{(\rho \to \mathcal{A}) \to \rho \to \mathcal{A}}\ (succ\ c\ x^\rho)\ x^\rho$ violates the criterion since *succ* is a modifier (produces $(x+1)^\rho$) and a free ρ-typed variable appears in its continuation. On the other hand, expression $\lambda c.\lambda x^\rho.\ chr^{(char \to \mathcal{A}) \to \rho \to \mathcal{A}}\ (succ\ c\ x^\rho)\ x^\rho$ satisfies the criterion since *chr* does not create a new ρ-typed value.

- Expression $C^{\rho \to \kappa} K^{\rho}$ is single-threaded if C is single-threaded, and if K is different from the last encountered ρ-value then no ρ-typed elements should appear in the stack (i.e. $nt_{\rho}(\kappa)$) and no free ρ-typed variables should occur in C (i.e. $nfv_{\rho}(C)$) (rule (Δ6)). For example, expressions *plusc fin* 1^{ρ} 2^{ρ} and $\lambda c.\lambda x^{\rho}$. *plusc c* x^{ρ} 1^{ρ} violate the criterion.
- The reduction of an operator yielding a ρ-value $opmc^{(\rho \to \nu m+1 \to \ldots \to \nu n \to \mathcal{A}) \to \kappa}$ $C\ V_1 \ldots V_m\ V_{m+1} \ldots V_n \xrightarrow[F]{1} C\ N^{\rho}\ V_{m+1} \ldots V_n$ entails that no element of type ρ should occur in $\{V_{m+1},\ldots,V_n\}$. (i.e. $nt_{\rho}(\nu_{m+1} \to \ldots \to \nu_n \to \mathcal{A})$ (rule (Δ2)). For example, expression $\lambda c.\lambda x^{\rho}$. $succ^{(\rho \to \rho \to \mathcal{A}) \to \rho \to \rho \to \mathcal{A}}$ *(plusc c)* x^{ρ} x^{ρ} violates the criterion (after reduction of *succ* there will be two different ρ-values (*x+1* and *x*) in the stack).

Two problems arise with closures:

- In general we do not know when the closure will be applied, so we enforce that no free ρ-typed variable occurs in closures.

- In an expression *v C* we do not know if variable *v* will be bound to a modifying expression or not. One solution is to find a sufficient condition *P* such that $P(\tau)$ implies that any closure of type τ is not a modifier of ρ. So, considering the expression $v^{\kappa 1 \to \kappa 2}$ *C*, if $P(\kappa 1 \to \kappa 2)$ then we can deduce that *v* will not be bound to a modifying expression and *C* can contain free variables of type ρ; otherwise no free variables of type ρ should occur in *C*. Here we choose, to be coherent with operator types, $P(\tau) = (\tau \not\equiv (\rho \to \kappa_1) \to \kappa_2)$ (Δ1) and *P* is made into a sufficient condition by rule (Δ8).

Recursive functions could have been treated in the same way as closures. However, the associated criterion can be less conservative because we know that the expression *rec f = V* will be bound to variable *f*. To this aim, recursive functions are assumed to bear different names. An expression r*ec f = V* is single-threaded if its body *V* is single-threaded assuming that variable *f* is single-threaded (rule (Δ5)). To check that a function does not modify ρ, the assumption is that *f* is not a modifier.

Our single-threading predicate Δ_{ρ}^{r} is recursively defined on the structure of expressions. We assume that type ρ is a first-order type and that primitive operators act on their arguments in a single-threaded fashion. Superscript r represents the last ρ-typed value encountered; initially r is set to a special value Ω different from any other value. Figure 3 gathers the criteria for single-threading.

Definition 2 *An expression E is said single-threaded in type* ρ *if* $\Delta_{\rho}^{\Omega}(E)$.

$\Delta_{\rho}^{r}(E^{\tau})$ iff:

(Δ1) $E^{\tau} \equiv fin^{\kappa} \vee c^{\kappa}$

(Δ2) $E^{\tau} \equiv (v^{\nu} \vee opmc^{\nu}) \wedge (\nu \equiv (\rho \to \kappa_1) \to \kappa_2 \Rightarrow nt_{\rho}(\kappa_1))$

(Δ3) $E^{\tau} \equiv ifc\ C_1\ C_2 \wedge \Delta_{\rho}^{r}(C_1) \wedge \Delta_{\rho}^{r}(C_2)$

(Δ4) $E^{\tau} \equiv \lambda x.F \wedge \Delta_{\rho}^{r}(F)$

(Δ5) $E^{\tau} \equiv rec\ f = V \wedge nfv_{\rho}(V) \wedge ((\Delta_{\rho}^{\Omega}(f) \vdash\!\!\!- \Delta_{\rho}^{\Omega}(V)) \vee (\Theta_{\rho}^{\Omega}(f) \vdash\!\!\!- \Theta_{\rho}^{\Omega}(V)))$

(Δ6) $E^{\tau} \equiv C^{\rho \to \kappa} K^{\rho} \wedge \Delta_{\rho}^{K}(C) \wedge (K \equiv r \vee (nfv_{\rho}(C) \wedge nt_{\rho}(\kappa)))$

(Δ7) $E^{\tau} \equiv C^{\rho \to \kappa} v^{\rho} \wedge \Delta_{\rho}^{r}(C)$

(Δ8) $E^{\tau} \equiv C^{\sigma \to \kappa} V^{\sigma} \wedge \Delta_{\rho}^{r}(C) \wedge nfv_{\rho}(V) \wedge$ if $\sigma \not\equiv (\rho \to \kappa_1) \to \kappa_2$ then $\Theta_{\rho}^{\Omega}(V)$ else $\Delta_{\rho}^{\Omega}(V)$

(Δ9) $E^{\tau} \equiv V^{\kappa 1 \to \kappa 2}\ C^{\kappa 1} \wedge ((\Theta_{\rho}^{r}(V) \wedge \Delta_{\rho}^{r}(C)) \vee (nfv_{\rho}(C) \wedge \Delta_{\rho}^{r}(V) \wedge \Delta_{\rho}^{\Omega}(C)))$

Figure 3 Single-threading criteria on continuation expressions (Predicate Δ)

Intuitively, Δ^K_ρ *(E)* means that *E* is single-threaded provided that the global variable implementing ρ-values has previously being initialized to *K* (and Δ^Ω_ρ *(E)* $\Rightarrow \forall K\ \Delta^K_\rho$ *(E))*. In the remainder of the paper, σ stands for a value type different from ρ. The non-modification criteria Θ^r_ρ *(E)* means that *E* is single-threaded and does not create new instances of type ρ. That is to say that *E* does not contain (except in closures of type $(\rho \to \kappa_1) \to \kappa_2$) expressions of the form $v^{(\rho \to \kappa 1) \to \kappa 2}\ C$, $opmc^{(\rho \to \kappa 1) \to \kappa 2}$ or $C\ K^\rho$ with $K \not\equiv r$. Θ^r_ρ is formally defined in Figure 4.

Θ^r_ρ (E) iff:

$E^\tau \equiv fin^\kappa \vee c^\kappa$

$E^\tau \equiv (v^\nu \vee opmc^\nu) \wedge \nu \not\equiv (\rho \to \kappa_1) \to \kappa_2$

$E^\tau \equiv ifc\ C_1\ C_2 \wedge \Theta^r_\rho\ (C_1) \wedge\ \Theta^r_\rho\ (C_2)$

$E^\tau \equiv \lambda x.F \wedge \Theta^r_\rho\ (F)$

$E^\tau \equiv rec\ f = V \wedge nfv_\rho(V) \wedge (\Theta^\Omega_\rho\ (f)\ \vdash \Theta^\Omega_\rho\ (V))$

$E^\tau \equiv C^{\rho \to \kappa}\ K^\rho \wedge \Theta^r_\rho\ (C) \wedge K \equiv r$

$E^\tau \equiv C^{\rho \to \kappa}\ v^\rho \wedge \Theta^r_\rho\ (C)$

$E^\tau \equiv C^{\sigma \to \kappa}\ V^\sigma \wedge \Theta^r_\rho\ (C) \wedge nfv_\rho(V) \wedge$ if $\sigma \not\equiv\ (\rho \to \kappa_1) \to \kappa_2$ then $\Theta^\Omega_\rho\ (V)$ else $\Delta^\Omega_\rho\ (V)$

$E^\tau \equiv V^{\kappa 1 \to \kappa 2}\ C^{\kappa 1} \wedge \Theta^r_\rho\ (V) \wedge \Theta^r_\rho\ (C)$

Figure 4 Non-modification criteria on continuation expressions (Predicate Θ)

We can now check by applying the Δ-rules that the function

rec f = λc.λa.λi.λp. eqc (ifc (c a) (accessc (multc (subc (updatec (f c) a i p) i 1) p) a i)) i 0

is single-threaded in ρ. As an example, we describe a few steps of the analysis of subexpression *subc (updatec (f c) a i p) i 1:*

Δ^Ω_ρ *(subc*$^{(\sigma \to \kappa 1) \to \kappa 2}$ *(…)* i^σ 1^σ*)*

$\Leftrightarrow \Delta^\Omega_\rho$ *(subc*$^{(\sigma \to \kappa 1) \to \kappa 2}$ *(…)* i^σ*)* $\wedge$ *nfv*$_\rho$*(*1^σ*)* $\wedge\ \Theta^\Omega_\rho$ *(1)* (Δ8)

nfv$_\rho$*(*1^σ*)* $\wedge\ \Theta^\Omega_\rho$ *(1)* is trivially true and the same applies to *i* thus

$\Leftrightarrow \Delta^\Omega_\rho$ *(subc*$^{(\sigma \to \kappa 1) \to \kappa 2}$ *(…))*

$\Leftrightarrow \Theta^\Omega_\rho$ *(subc*$^{(\sigma \to \kappa 1) \to \kappa 2}$*)* $\wedge\ \Delta^\Omega_\rho$ *(updatec (f c)* a^ρ *i p)* (Δ9)

and so on.

One important property of Δ is that it is preserved by the reduction by First.

Property 3 $(\forall\ E^{\mathcal{A}})\ \Delta^r_\rho\ (E) \wedge E\ \xrightarrow[F]{*} F \Rightarrow \Delta^r_\rho\ (F)$

Proof: Showing $\Delta^r_\rho\ (E) \wedge E\ \xrightarrow[F]{1} F \Rightarrow \Delta^r_\rho\ (F)$ for each rule of First. The only tedious part is the proof that the predicate is preserved by β-reduction; the following lemmas (each one is proved by structural induction) resolve this point:

- $\Delta^r_\rho\ ((\lambda c.C_1)\ C_2) \Rightarrow \Delta^r_\rho\ (C_1[C_2/c])$
- $\Delta^K_\rho\ ((\lambda v.C)\ K^\rho) \Rightarrow \Delta^K_\rho\ (C[K/v])$ *and* $\Theta^K_\rho\ ((\lambda v.C)K) \Rightarrow \Theta^K_\rho\ (C[K/v])$
- $\Delta^r_\rho\ ((\lambda v.C)\ V^\sigma) \Rightarrow \Delta^r_\rho\ (C[V/v])$ *and* $\Theta^r_\rho\ ((\lambda v.C)V) \Rightarrow \Theta^r_\rho\ (C[V/v])$

We tried to keep the criteria from being too complex and did not mention some less conservative but more intricate options. One potential source of failure lies in our treatment of closures. Our requirement (if $\sigma \not\equiv\ (\rho \to \kappa_1) \to \kappa_2$ then $\Theta^\Omega_\rho\ (V)$ else $\Delta^\Omega_\rho\ (V)$) is somewhat simplistic and better solu-

tions could have been chosen. The best way would be to perform a closure analysis to detect the set of closures that each higher-order variable *v* may be bound to; non-modification has to be enforced on those closures only if really needed (for example if expression *v C* has free ρ-typed variables). This may be too costly an analysis and one might prefer alternative solutions relying on types. For each closure V^{σ} we might check $\Delta_{\rho}^{r}(V) \wedge \neg\Theta_{\rho}^{r}(V)$ constructing this way a set of modifying closure types $\mathcal{M}$ (i.e. $\sigma \in \mathcal{M}$ iff there is a closure *V* of type σ such that $\Delta_{\rho}^{r}(V) \wedge \neg\Theta_{\rho}^{r}(V)$). An expression v^{ν} *C* with free ρ-typed variables could be single-threaded only if type ν does not belong to $\mathcal{M}$. Another case of failure occurs when a free ρ-typed variable is enclosed. There are ways to relax this restriction. For instance, when the closure application time (or a safe approximation of it) is known, it is sufficient to enforce that no modification occurs until then.

3.2 Global variable transformation

Property 3 does not state that ρ can indeed be implemented by a global variable. In order to show that globalization can be performed on single-threaded expressions we now give a formal specification of this optimization. Global variable transformation has been expressed using a Simula-like class [19] or by adding explicit assignments [18]. We choose here to stay in the functional framework and to describe the globalization in terms of program transformations. The transformation produces expressions of the form $V\ C\ R\ V_1 \ldots V_n$ where the extra argument *R* plays the role of a global variable. Operationally, expressions can still be seen as a stack machine state where *R* denotes a register. Transformation $\mathcal{R}$ (Figure 5) removes variables of type ρ (rules *(R5),(R8)*) and replaces the creation of new ρ-values by destructive updates (rules *(R3),(R7)*). Applications are transformed to take into account the global variable (rules *(R10), (R11)*). If the global expression yields a result of type σ (different from ρ) then continuation *fin* removes the global variable before returning the answer (rule *(R2)*). Destructive versions of operators are introduced; for example:

$$plusd\ C\ R\ V \rightarrow_{\delta} C\ (R+V)$$

$$updated\ C\ [a_1,\ldots a_i,\ldots a_m]\ i\ V \rightarrow_{\delta} C\ [a_1,\ldots,a_{i-1},V,\ldots a_m]$$

More generally, operators *opmc* such that $opmc\ C\ V_1^{\sigma 1} \ldots V_i^{\rho} \ldots\ V_m^{\sigma m} \rightarrow_{\delta} C\ N^{\sigma}$ are transformed into operators *opmd* such that $opmd\ C\ V_i^{\rho}\ V_1^{\sigma 1} \ldots V_{i-1} V_{i+1} \ldots\ V_m^{\sigma m} \rightarrow_{\delta} C\ V_i^{\rho}\ N^{\sigma}$. Modifiers *opmc* such that $opmc\ C\ V_1^{\sigma 1} \ldots V_i^{\rho} \ldots\ V_m^{\sigma m} \rightarrow_{\delta} C\ N^{\rho}$ are transformed into operators *opmd* such that $opmd\ C\ V_i^{\rho}\ V_1^{\sigma 1} \ldots V_{i-1} V_{i+1} \ldots\ V_m^{\sigma m} \rightarrow_{\delta} C\ N^{\rho}$. That is to say, if *opmc* takes *m* arguments, *n* being ρ-typed, *opmd* takes only *(m-n)+1* arguments (single-threading enforces the *n* ρ-typed elements to be equal) and yields the same value as *opmc*, updating the global variable if the result is of type ρ (rule *(R3)*). The conditional is treated in the same way *(R4)*

(R1) $\mathcal{R}_{\rho}(a) = a$ if $a \equiv v, c, K$ or $fin^{\rho \rightarrow A}$

(R2) $\mathcal{R}_{\rho}(fin^{\sigma \rightarrow A}) = \lambda r.\ fin$

(R3) $\mathcal{R}_{\rho}(opc) = opd$

(R4) $\mathcal{R}_{\rho}(ifc\ C_1\ C_2) = ifd\ \mathcal{R}_{\rho}(C_1)\ \mathcal{R}_{\rho}(C_2)$

(R5) $\mathcal{R}_{\rho}(\lambda v^{\rho}.\ C) = \mathcal{R}_{\rho}(C)$

(R6) $\mathcal{R}_{\rho}(\lambda v^{\sigma}.\ C) = \lambda r.\lambda v.\mathcal{R}_{\rho}(C)\ r$

(R7) $\mathcal{R}_{\rho}(E\ K^{\rho}) = \lambda r.\mathcal{R}_{\rho}(E)\ K$

(R8) $\mathcal{R}_{\rho}(C\ v^{\rho}) = \mathcal{R}_{\rho}(C)$

(R9) $\mathcal{R}_{\rho}(\lambda c.\ V) = \lambda c.\mathcal{R}_{\rho}(V)$

(R10) $\mathcal{R}_{\rho}(V\ C^{\kappa}) = \mathcal{R}_{\rho}(V)\ \mathcal{R}_{\rho}(C)$

(R11) $\mathcal{R}_{\rho}(C\ V^{\sigma}) = \lambda r.\ \mathcal{R}_{\rho}(C)\ r\ \mathcal{R}_{\rho}(V)$ or using a convenient combinator = $push\ \mathcal{R}_{\rho}(V)\ \mathcal{R}_{\rho}(C)\ (push\ V\ C\ R \underset{F}{\rightarrow} C\ R\ V)$

Figure 5 Global Variable Transformation $\mathcal{R}$

The expressions produced by this transformation contain at most one free occurrence of ρ-typed variables. Operationally, this means that no ρ-value will ever be pushed on the stack. The correctness property (Property 4) states that if an expression $E^{\mathcal{A}}$ is single-threaded then its reduction and the reduction of its transformed version produce equivalent results.

Property 4 *For all closed $E^{\mathcal{A}}$ such that $\Delta^{\Omega}_{\rho}(E)$, if E has a whnf, i.e. $E \xrightarrow[F]{*} fin\ N$ then $(\forall R^{\rho})\ \mathcal{R}_{\rho}(E)\ R \xrightarrow[F]{*} fin\ \mathcal{R}_{\rho}(N)$*

Proof: Induction requires a stronger property that we do not describe here for the sake of brevity. The structure of the proof is as before by showing the property for one reduction step using lemmas $(\mathcal{R}_{\rho}(E)[\mathcal{R}_{\rho}(C^{\kappa})/c] \equiv \mathcal{R}_{\rho}(E[C^{\kappa}/c]),\ \mathcal{R}_{\rho}(E)[\mathcal{R}_{\rho}(V^{\sigma})/v] \equiv \mathcal{R}_{\rho}(E[V^{\sigma}/v])$ which are themselves proved by structural induction; Property 3 is then used for the induction on the length of the reduction.

Example: The application of the global variable transformation to function *f*

rec f = λc.λa.λi.λp. eqc (ifc (c a) (accessc (multc (subc (updatec (f c) a i p) i 1) p) a i)) i 0

yields

rec f = λc.λr.λi.λp. push 0 (push i (eqd (ifd (c) (push i (accessd (push p (multd (push 1 (push i (subd

(push p (push i (updated (f c))))))))))))) r

The array argument has been replaced by the global variable *r*. We describe below some of the reduction steps of the application of this function to arguments *fin*, *[4,3,2]*$^{\rho}$, *3*, *1*.

(rec f = λc.λr.λi.λp. push 0 (push i (eqd (ifd (c) (push i (...)))) r) fin [4,3,2] 3 1

$\xrightarrow[F]{}$ *push 0 (push 3 (eqd (ifd (fin) (push 3 (...)))) [4,3,2]*

$\xrightarrow[F]{}$ *push 3 (eqd (ifd (fin) (push 3 (...)))) [4,3,2] 0*

$\xrightarrow[F]{}$ *....*$\xrightarrow[F]{}$ *updated ((rec f =...) fin) [4,3,2] 3 1 2 2* {destructive update of the 3rd element}

$\xrightarrow[F]{}$ *(rec f =λc.λr.λi.λp....) fin [4,3,1] 2 2*

$\xrightarrow[F]{}$ *....*$\xrightarrow[F]{}$ *updated (f fin) [4,3,1] 2 2 1 6* {destructive update of the 2nd element}

$\xrightarrow[F]{}$ *(rec f =λc.λr.λi.λp....) fin [4,2,1] 1 6* $\xrightarrow[F]{}$ *....*

$\xrightarrow[F]{}$ *updated ((rec f =...)fin) [4,2,1] 1 6 0 24* {destructive update of the 1st element}

$\xrightarrow[F]{}$ *(rec f =λc.λr.λi.λp....) fin [6,2,1] 0 24*

$\xrightarrow[F]{}$ *...* $\xrightarrow[F]{}$ *ifd (fin) (push 0 (...)) [6,2,1] True* $\xrightarrow[F]{}$ *fin [6,2,1]*

Criteria Δ and transformation $\mathcal{R}$ extend trivially to single-threading in several types $\rho_1,...\rho_i$. The transformed expressions would have several explicit global variables and would be of the form *V C* $R_1 ... R_i\ V_1 ... V_n$.

4 Applications

In this section we describe several applications of the analysis: the compilation of functional languages (section 4.1), semantics-directed compilation (section 4.2) and globalization-based transformations (section 4.3).

4.1 Compilation of functional languages

The primarily goal of our work was to improve the compilation of functional programs. We are currently working on the integration of our analysis in a transformation-based compiler [5,6]. The first step of this compiler is the compilation of the computation rule; call-by-value, call-by-need and even mixed evaluation orders (e.g. call-by-need with strictness information) are compiled using CPS transformations; single-threading is then detected on the resulting continuation expressions. Preliminary results indicate that this optimization is very effective for iterative functions where the use of registers can reduce execution time up to *50* per cent and, of course, for the implementation of arrays. Our criteria is also helpful for non single-threaded functions since they detect single-threaded subexpressions which can be locally transformed by $\mathcal{R}$.

However, there are cases where a standard type annotation would not fit well with the analysis. For example, when two parameters have the same type, we would try to detect if both of them can be implemented using a single global variable. In most cases this would fail whereas they can be implemented in two different variables. Let *itfact* be the iterative version of the factorial function:

$rec\ itfact = \lambda c^{int \to \mathcal{A}}.\lambda x^{int}.\lambda y^{int}.\ eqc\ (ifc\ (c\ y)\ (multc\ (subc\ (itfact\ c)\ x\ 1)\ x\ y))\ 0\ x$

Δ_{int} fails because *itfact* is obviously not single-threaded in type *int*.

We are primarily interested in globalizing parameters of recursive functions, a simple solution consists in annotating types so that each parameter has a different type. A recursive function of type $(\kappa \to \mathcal{A}) \to \nu \to \nu \to \mathcal{A}$ is typed $(\kappa \to \mathcal{A}) \to \nu 1 \to \nu 2 \to \mathcal{A}$. Type inference will be performed using this information and assuming that τi matches the unannotated type τ (e.g. $int1 \to int$ and $int \to int2$ are unified into $int1 \to int2$).

If the factorial function is given type $(int \to \mathcal{A}) \to int1 \to int2 \to \mathcal{A}$ then type inference produces new types for operators and the type of *itfact* becomes $(int2 \to \mathcal{A}) \to int1 \to int2 \to \mathcal{A}$.

$rec\ itfact = \lambda c^{int2 \to \mathcal{A}}.\lambda x^{\ int1}.\lambda y^{\ int2}.\ eqc\ (ifc\ (c\ y^{int2})$

$(multc^{(int2 \to \mathcal{A}) \to int1 \to int2 \to \mathcal{A}}(subc^{(int1 \to int2 \to \mathcal{A}) \to int1 \to int \to int2 \to \mathcal{A}}$

$(itfact^{(int2 \to \mathcal{A}) \to int1 \to int2 \to \mathcal{A}}\ c)\ x^{int1}\ 1)\ x^{int1}\ y^{int2}))\ 0\ x^{\ int1}$

Using this annotation, Δ_{int1} and Δ_{int2} detect that *itfact* is single-threaded in types *int1* and *int2*.

Type inference must be extended a bit to be able to type functions like $rec\ f = \lambda c.\lambda x^{\ int}.\lambda y^{\ int}$... $subc\ (f\ c\ y)\ y\ 1$... If f has type $(int \to \mathcal{A}) \to int1 \to int2 \to \mathcal{A}$, then $f\ c$ y can not be typed. To solve this point, an operator *Idc*, with reduction rule $Idc\ C\ V \to C\ V$, is introduced. The previous expression becomes $Idc\ (f\ c)\ y$ and function f can now be typed:

$rec\ f = \lambda c.\lambda x^{\ int1}.\lambda y^{\ int2}....\ subc\ (Idc^{(int1 \to int2 \to \mathcal{A}) \to int2 \to int2 \to \mathcal{A}}\ (f\ c)\ y^{int2})\ y^{\ int2}\ 1)$...

If a function is single threaded in type ρ_1 and ρ_2 then an operator $Idc^{(\rho 1 \to \kappa) \to \rho 2 \to \kappa}$ corresponds to the affectation $R_1 := R_2$, R_1 and R_2 being the global variables implementing ρ_1 and ρ_2.

This heuristic for annotating types fits our needs but is not optimal in that it would fail to detect when several arguments can indeed be implemented by a single global variable. As mentioned in section 6.2, a better solution would be to use the information provided by the single-threading criteria to choose the annotation.

4.2 Semantics-directed compiler generation

We plan to integrate our analysis into a semantics-directed compiler generator which would also include a CPS transformation. Starting from a direct denotational semantics, a first step is the defunctionalization of the store introducing a first-order data structure (e.g. an array). The second step is the compilation of the computation rule using a CPS transformation. Call-by-value is chosen if the semantics uses a strict λ-calculus or if we can infer (using a strictness analysis) that the valuation functions are strict; otherwise call-by-need is used. Single-threadedness of the store domain is then analyzed on the resulting continuation expressions and the globalization transformation is applied if possible. The next step is the partial evaluation of the semantics function applied to a particular program. Let us take an example to illustrate these steps; let $\mathcal{P}$ be a specification of a small imperative language (cf. Figure 6). The valuation functions are assumed to be strict in their arguments.

$\mathcal{P}$: Program $\to$ Store $\to$ Store

$\mathcal{P}[\![\text{C.}]\!] = \mathcal{C}[\![\text{C}]\!]$

$\mathcal{C}$: Command $\to$ Store $\to$ Store

$\mathcal{C}[\![\text{I:=E}]\!] = \lambda s.\ \text{update s } [\![\text{I}]\!] (\mathcal{E}[\![\text{E}]\!]\ s)$

$\mathcal{C}[\![\text{C}_1\text{;C}_2]\!] = \lambda s.\ \mathcal{C}[\![\text{C}_2]\!] (\mathcal{C}[\![\text{C}_1]\!]\ s)$

$\mathcal{C}[\![\textbf{if B then } \text{C}_1 \textbf{ else } \text{C}_2]\!] = \lambda s.\ \text{if } (\mathcal{B}\ [\![\text{B}]\!]\ s)\ (\mathcal{C}\ [\![\text{C}_1]\!]\ s)\ (\mathcal{C}[\![\text{C}_2]\!]\ s)$

$\mathcal{C}[\![\textbf{while B do C}]\!] = (\text{rec loop} = \lambda s.\ \text{if } (\mathcal{B}\ [\![\text{B}]\!]\ s)\ (\text{loop } (\mathcal{C}\ [\![\text{C}]\!]\ s))\ s)$

$\mathcal{E}$: Expression $\to$ Store $\to$ Int

$\mathcal{E}[\![\text{E}_1 * \text{E}_2]\!] = \lambda s.\ \text{mult } (\mathcal{E}[\![\text{E}_1]\!]\ s)\ (\mathcal{E}[\![\text{E}_2]\!]\ s)$ {same thing for E_1 - E_2, E_1 + E_2,...}

$\mathcal{E}[\![\text{I}]\!] = \lambda s.\ \text{access s } [\![\text{I}]\!]$

$\mathcal{E}[\![\text{N}]\!] = \lambda s.\ \mathcal{N}[\![\text{N}]\!]$

$\mathcal{B}$: Expression $\to$ Store $\to$ Bool (omitted)

Figure 6 Denotational Semantics of a Small Imperative Language (extract)

These functions are first transformed using a CPS conversion (e.g. $\mathcal{V}$ described in section 5). For instance, the first equation describing commands becomes $\lambda c.\lambda s.\ \mathcal{E}[\![E]\!]\ (updatec\ c\ s\ [\![I]\!])s$. Predicate Δ_{Store} is then applied to check that command equations are single-threaded in their store domain. For the equation above, we have to check $\Theta_{Store}(\mathcal{E}[\![E]\!])$ since this expression has a free store-typed variable in its continuation. Transformation $\mathcal{R}$ is applied and the first command equation is now $\lambda c.\lambda r.\ \mathcal{E}[\![E]\!]\ (\lambda r.\ push\ [\![I]\!]\ (updated\ c)\ r)\ r$; the global variable r is suppressed by η-reduction and we get $\lambda c.\mathcal{E}[\![E]\!]\ (push\ [\![I]\!]\ (updated\ c))$. Figure 7 gathers the transformed equations after globalization.

$\mathcal{P}$: Program → Cmdcont → Cmdcont

$\mathcal{P}$⟦C.⟧ = $\mathcal{C}$⟦C⟧

$\mathcal{C}$: Command → Cmdcont → Cmdcont

$\mathcal{C}$⟦I:=E⟧ = λc. $\mathcal{E}$⟦E⟧ (updated $_{[I]}$ c) (updated $_{[I]}$ being an abbreviation for push [[I]] o updated)

$\mathcal{C}$⟦C_1;C_2⟧ = λc. $\mathcal{C}$⟦C_1⟧ ($\mathcal{C}$⟦C_2⟧ c)

$\mathcal{C}$⟦**if** B **then** C_1 **else** C_2⟧ = λc. $\mathcal{B}$ ⟦B⟧ (ifc ($\mathcal{C}$ ⟦C_1⟧ c) ($\mathcal{C}$⟦C_2⟧ c))

$\mathcal{C}$⟦**while** B **do** C⟧ = (rec loop = λc. $\mathcal{B}$ ⟦B⟧ (ifc ($\mathcal{C}$ ⟦C_1⟧ (loop c)) c))

$\mathcal{E}$: Expression → ExprCont → Cmdcont

$\mathcal{E}$⟦E_1 * E_2⟧ = λc. $\mathcal{E}$⟦E_2⟧ ($\mathcal{E}$⟦E_1⟧ (multd c))

$\mathcal{E}$⟦I⟧ = λc. accessd $_{[I]}$ c

$\mathcal{E}$⟦N⟧ = push $\mathcal{N}$⟦N⟧

$\mathcal{B}$: Expression → ExprCont → Cmdcont (omitted)

Figure 7 Semantics after CPS and globalization transformations

This semantics applied to a program is simplified by partial evaluation. For that particular semantics, this step is straightforward since it amounts to β-reduce continuations. The expressions obtained can be seen as generic stack machine code. For example:

$\mathcal{P}$[[X:=2;Y:=Y*X.]] = λc. $\mathcal{C}$[[X:=2]] ($\mathcal{C}$[[Y:=Y*X]] c) =...

= λc. push 2 (updated $_{[[X]]}$ (accessd $_{[[X]]}$ (accessd $_{[[Y]]}$ (multd (updated $_{[[Y]]}$ c)))))

This method produces very good quality code for such toy languages. This would not be the case if environments and procedures were added to the language. Much work remains to be done in order to automatically derive efficient compilers for real life languages.

4.3 Globalization-directed transformations

We are interested in this section in non single-threaded functions. No extension of the criteria could help, however our analysis returns information which can be turned to account to transform non single-threaded functions into single-threaded ones. One of the most common cases of non single-threadedness is when arguments are not evaluated in a proper order. The evaluation strategy enforces a specific order which may invalidate single-threading, whereas arguments of strict functions can be evaluated in any order. In order to deal with this problem, we define program transformations which can be seen as local modifications of the evaluation ordering. Let us come back to the *f* function defined in section 3.

rec f = λc.λa.λi.λp. eqc (ifc (c a) (accessc (multc (subc (updatec (f c) a i p) i 1) p) a i)) i 0

Function *f* is not single-threaded in its third argument which is modified while still referenced. Let ρ be the type of the third argument; Δ_ρ points out that *f* is not single-threaded in ρ because there are occurrences of ρ-typed free variable *i* in the continuation of modifier *subc*. The idea is to postpone the evaluation of *subc* until it can be done in a single-threaded fashion or it becomes necessary.

Let Π be a function such that:

$\Pi_i(F^\nu) = j \Rightarrow (\forall C,V_1,...,V_i)\ (F\ C\ V_1 ...\ V_i \xrightarrow[F]{*} C\ U_1 ...\ U_{j-1} V_i) \vee (F\ C\ V_1 ...\ V_i = \perp)$

Intuitively, this function indicates that the *ith* element of the stack is not needed by *F* and will be at position *j* in the stack after the execution of *F*. Of course, Π is a partial function which yields only a safe approximation ($\perp$ or an integer). We do not describe it here and we just give the rule that is needed for our example:

$\Pi_i(opmc) = i\text{-}m+1$ *if* $i>m$ otherwise $\perp$

Using this function, safe adjustments of the evaluation ordering can be performed. One basic transformation is:

$F\ (E\ C\ V_1 \ldots V_p) \leftrightarrow E\ (\lambda v_1.\ \ldots \lambda v_{n-1}.\ F\ (C\ v_1 \ldots v_{n-1}))\ V_1 \ldots V_p \qquad \text{if } \Pi_{p+1}(E) = n > 0$

with $p \geq 0$ and $v_1,\ldots,v_{n-1}$ being fresh variables.

Evaluations of *F* and *E* are inverted and it is easy to prove that both expressions are operationally equivalent. This transformation is worth applying when non single-threading comes from free variables in the continuation of a modifier, that is when $\neg\Theta_\rho(F) \wedge fv_\rho(C)$ is detected. The above transformation is used to delay the reduction of the modifier until it is needed or, hopefully, until the free variables are eventually consumed.

When analyzing the subexpression *subc (updatec (f c) a i p) i 1*, we get Δ_ρ *(subc)* $\wedge$ fv_ρ*(updatec (f c) a i p)* and $\Pi_4(updatec) = 2$, thus the transformation can be applied and returns *updatec* ($\lambda v.$ *subc (f c v)) a i p i 1*. Function *f* is now single-threaded in its second and third arguments.

5 Single-threading detection on source expressions

All the results described so far apply to continuation expressions. It is not however compulsory and we describe in this section how to use Δ to get criteria on regular λ-expressions according to different computation rules. The idea is to design a CPS conversion $\mathcal{C}$ mapping source expressions to continuation expressions according to a specific evaluation scheme $\mathcal{E}$. Transformation $\mathcal{C}$ must produce expressions whose reduction by First models the reduction by $\mathcal{E}$ of the original expressions. By simplifying Δ o $\mathcal{C}$ one can get single-threading criteria on source expressions for the evaluation ordering $\mathcal{E}$. Figure 8 describes a transformation compiling left-to-right call-by-value. For the sake of brevity, we consider here a λ-calculus extended with constants and operators only.

- $\mathcal{V}(v^\tau) = \lambda c.c^{\overline{\tau} \to \mathcal{A}}\ v^{\overline{\tau}}$
- $\mathcal{V}(op^\tau) = \lambda c.c^{\overline{\tau} \to \mathcal{A}}\ opc^{\overline{\tau}}$
- $\mathcal{V}(\lambda v^{\tau 1}.F^{\tau 2}) = \lambda c.c^{(\overline{\tau 1 \to \tau 2}) \to \mathcal{A}}\ (\lambda c.\lambda v^{\overline{\tau 1}}.\mathcal{V}(F)\ c^{\overline{\tau 2} \to \mathcal{A}})$
- $\mathcal{V}(E_1{}^{\tau 1 \to \tau 2}\ E_2{}^{\tau 1}) = \lambda c^{\overline{\tau 2} \to \mathcal{A}}.\mathcal{V}(E_1)\ (\lambda f^{(\overline{\tau 2} \to \mathcal{A}) \to \overline{\tau 1} \to \mathcal{A}}.\mathcal{V}(E_2)\ (f\ c))$

With the associated transformation on types defined by: $\overline{\beta} = \beta$ and $\overline{\tau 1 \to \tau 2} = (\overline{\tau 2} \to \mathcal{A}) \to \overline{\tau 1} \to \mathcal{A}$

Figure 8 CPS transformation for left-to-right call-by-value ($\mathcal{V}$)

The proof that $\mathcal{V}$ models properly left-to-right call-by-value is beyond the scope of this paper. The interested reader may refer to [6] which presents such a proof.

We proceed by simplifying Δ o $\mathcal{V}$ to get criteria defined on the syntax of source expressions. We do not describe the simplification process which is straightforward; for example, the first rule (if *E* is an atom then *E* is single-threaded) follows from: Δ_ρ o $\mathcal{V}(v) = \Delta(\lambda c.c^{\overline{\tau} \to \mathcal{A}}\ v^{\overline{\tau}}) =$ *True* and Δ o $\mathcal{V}(op^\tau) = \Delta(\lambda c.c^{\overline{\tau} \to \mathcal{A}}\ opc^{\overline{\tau}}) =$ *True* (since $nt_\rho(\mathcal{A})$). For all expression *E*, $(\Theta_\rho$ o $\mathcal{V})$ *E* can be shown equivalent

to: $(\Delta_\rho \circ \mathcal{V})\ E \wedge$ (*E does not contain active expressions of type* ρ *different from an identifier*), where an expression is said *active* if it is not properly contained within a λ-abstraction. The resulting predicate is described in Figure 9.

$\Delta_\rho \circ \mathcal{V}(E)$ iff:

- $E \equiv$ atom (v or op)
- $E \equiv \lambda v^{\tau 1}.F^{\tau 2} \wedge \Delta_\rho \circ \mathcal{V}(F) \wedge$
  - (i) $\tau 1 \equiv \rho \Rightarrow$ all free ρ-typed identifiers in F are v^ρ
  - (ii) $\tau_1 \not\equiv \rho \Rightarrow$ F has no active ρ-typed expressions
  - (iii) $\tau_2 \not\equiv \rho \Rightarrow$ E does not contain active expressions of type ρ different from an identifier.
- $E \equiv E_1{}^{\tau 1 \to \tau 2}\ E_2{}^{\tau 1} \wedge \Delta_\rho \circ \mathcal{V}(E_1) \wedge \Delta_\rho \circ \mathcal{V}(E_2) \wedge$

  $fv_\rho(E_2) \Rightarrow$ all occurrences of active ρ-typed expressions in E_2 are occurrences of identifiers.

Figure 9 Single-threading criteria for λ−calculus using left-to-right cbv ($\Delta_\rho \circ \mathcal{V}$)

If right-to-left call-by-value is used, the associated CPS conversion $\mathcal{V}_r'$ remains the same except for the application rule which becomes:

$$\mathcal{V}_r'(E_1\ E_2) = \lambda c^{\overline{\tau 2} \to \mathcal{A}}.\mathcal{V}(E_2)\ (\lambda a^{\overline{\tau 1}}.\mathcal{V}(E_1)\ (\lambda f.f^{(\overline{\tau 2} \to \mathcal{A}) \to \overline{\tau 1} \to \mathcal{A}}\ c\ a))$$

The criteria remain the same except for the last one which becomes:

- $E \equiv E_1{}^{\tau 1 \to \tau 2}\ E_2{}^{\tau 1} \wedge \Delta_\rho \circ \mathcal{V}_r'(E_1) \wedge \Delta_\rho \circ \mathcal{V}_r'(E_2) \wedge$
  - (i) $\tau_1 \equiv \rho \Rightarrow$ all occurrences of active ρ-typed expressions in E_1 are occurrences of identifiers
  - (ii) $fv_\rho(E_1) \Rightarrow$ all occurrences of active ρ-typed expressions in E_2 are occurrences of identifiers.

In order to introduce as few continuations as possible, efficient transformations often use the following optimizations [6]:

- $\mathcal{V}(\text{opm } V_1 \ldots V_m) = \lambda c.\ \mathcal{V}(V_m)\ (\ldots\ (\mathcal{V}(V_1)\ (\text{opmc c}))\ldots)$
- $\mathcal{V}((\lambda v_1.\ \ldots \lambda v_n.\ F_1)\ F_2 \ldots F_n) = \lambda c.\ \mathcal{V}(F_n)\ (\ldots\ (\mathcal{V}(F_2)\ (\lambda v_1.\ \ldots \lambda v_n.\ \mathcal{V}(F_1)\ c))\ldots)$

This amounts to introducing multi-applications in the language and single-threading detection takes a great benefit from it. For example, the expression $\lambda x^\rho.\lambda y.x$ would not be single-threaded in general: it can be applied to one argument and yields a closure $\lambda y.x[K^\rho/x]$ containing ρ-typed elements (and one can check that $(\Delta_\rho \circ \mathcal{V})\lambda x^\rho.\lambda y.x$ = *False*). This would have drastic effects on single-threading since only the last parameter of functions would be a candidate for globalization. However, if the expression is known to be fully applied, no closure has to be built. Taking into account the above optimizations, we get an improved version of our criteria. For example:

- $E \equiv (\lambda v_1.\ \ldots \lambda v_n.\ F_1)\ F_2 \ldots F_n \wedge \Delta_\rho \circ \mathcal{V}(F_1) \wedge \ldots . \wedge \Delta_\rho \circ \mathcal{V}(F_n) \wedge$

$fv_\rho(F_i) \Rightarrow$ all occurrences of active ρ-typed expressions in $F_{i-1},\ldots,F_{1,}$ are occurrences of identifiers.

Using this criterion, the expression $\lambda x^\rho.\lambda y.x$ applied to two arguments is detected as single-threaded

It would also be possible to get criteria for call-by-need but they would be of little use since most parameters would be enclosed, and few globalizable variables would be found. However, a call-by-need with strictness information can benefit from single-threading analysis. Let us assume

that $\underline{E_1}\ E_2$ indicates that E_1 is strict and $\underline{v}$ means that v is defined by a strict λ-abstraction $\lambda v.E$. Then a CPS transformation for call-by-need can make use of these pieces of information in order to evaluate arguments of strict functions before calling them (cf. Figure 10).

- $\mathcal{N}(\!|v|\!) = v$
- $\mathcal{N}(\!|\underline{v}|\!) = \lambda c.c\ v$ {v is already evaluated}
- $\mathcal{N}(\!|op|\!) = \lambda c.c\ opc$
- $\mathcal{N}(\!|\lambda v.F|\!) = \lambda c.c\ (\lambda c.\lambda v.\mathcal{N}(\!|F|\!)\ c)$
- $\mathcal{N}(\!|E_1\ E_2|\!) = \lambda c.\mathcal{N}(\!|E_1|\!)\ (\lambda f.f\ c\ \mathcal{N}(\!|E_2|\!))$
- $\mathcal{N}(\!|\underline{E_1}\ E_2|\!) = \lambda c.\mathcal{N}(\!|E_2|\!)\ (\lambda a.\mathcal{N}(\!|E_1|\!)\ (\lambda f.f\ c\ a))$ {E_1 is strict; its argument is evaluated}

Figure 10 CPS transformation for call-by-need with strictness annotations ($\mathcal{N}$)

Therefore, by simplifying $\Delta \circ \mathcal{N}$, we can get criteria for call-by-need with strictness annotations.

This method of deriving criteria on source expressions is general (at least for sequential computation rules) and quite simple. However, to insure its correctness one has to check that the CPS transformation models the underlying implementation properly; in particular, the sequencing of variable definitions and uses must be identical.

6 Conclusions

We have presented sufficient syntactic criteria to detect the single-threading property on both continuation expressions and standard λ-expressions. Syntactic analysis is attractive by its low cost and by the information it provides. The approach is general enough to be applied to the compilation of functional languages, semantics-directed compiler generation and goal-directed transformations. Our work improves upon previous solutions (both syntactic and semantics-based) in that it applies to higher-order languages and to most sequential evaluation strategies.

6.1 Related Works

The closest related work is Schmidt's [19] which performs a syntactic analysis on a λ-calculus with a call-by-value semantics. His criteria can be described as the predicate $\mathcal{S}_\rho$ in Figure 11.

S_ρ iff:

- $E \equiv$ atom (v or op)
- $E \equiv \lambda v^\tau.F \wedge \mathcal{S}_\rho(F) \wedge$
  - (i) $\tau \equiv \rho \Rightarrow$ all free ρ-typed identifiers in F are v^ρ
  - (ii) $\tau \not\equiv \rho \Rightarrow$ F has no active ρ-typed expressions
- $E \equiv (E_1\ E_2)^\tau \wedge \mathcal{S}_\rho(E_1) \wedge \mathcal{S}_\rho(E_2) \wedge$
  - (i) $\tau \equiv \rho \Rightarrow$ if both E_1 and E_2 contain one or more active ρ-typed expressions then all of the active ρ-typed expressions in E are occurrences of the same identifier.
  - (ii) $\tau \not\equiv \rho \Rightarrow$ all occurrences of active ρ-typed expressions in E are occurrences of the same identifier

Figure 11 Schmidt's single-threading criteria (call-by-value)

Our criteria combined with transformations $\mathcal{V}$ or $\mathcal{V}_r$ turns out to be less conservative than $\mathcal{S}$. Formally stated:

Property 5 $\mathcal{S}_\rho(E) \Rightarrow (\Delta^\Omega_\rho \circ \mathcal{V}(E)) \wedge (\Delta^\Omega_\rho \circ \mathcal{V}_r(E))$

This property is easily shown by structural induction using definitions of $\Delta_\rho \circ \mathcal{V}$ and $\Delta_\rho \circ \mathcal{V}_r$ (section 5). Actually, Property 5 would hold for any transformation $\mathcal{V}$ compiling a call-by-value reduction strategy. The main reason is that $\mathcal{S}_\rho$ does not rely upon a particular version of call-by-value. $\mathcal{S}_\rho$ is valid for any sequential or parallel version of call-by-value and therefore does not take advantage of a particular evaluation order. If this loss of information may be accepted in semantics-directed compiling (where a language designer may make conscious use of the criteria), it would spoil many optimization opportunities when used in a functional language compiler. Another drawback of S_ρ is that multi-applications (cf. Section 5) are not considered and it detects only functions that are single-threaded in their last parameter. This work has been extended in order to analyse combinator languages [15].

Most of the other single-threading detection methods are based on abstract interpretation with some kind of operational semantics describing the sequencing of definitions and uses of variables. [13] builds a lifetime grammar to detect global variables in procedural programs. Bloss [2] uses the notion of path semantics to analyze a lazy first order functional language. [3] considers the same kind of language but accepts call-by-need with strictness information; this analysis, while still in exponential time, is faster than Bloss'. Sestoft [18] focuses on function parameters and considers a strict higher order functional language. The single-threading criteria are defined by an interference analysis on a definition-use grammar constructed from a path semantics. This approach is quite different from ours and a formal assessment of their relative power is difficult to achieve. However, since Sestoft's analysis includes a closure analysis it might detect globalizable parameters that we fail to detect. On the other hand, multi-applications are not considered and this analysis has the same shortcoming as Schmidt's. A notable difference is that Sestoft's work does not rely on types and therefore is well-suited for untyped languages. This work has been extended recently in order to detect globalizable parameters that may be captured in closures [7]. It is worth mentioning that they also found the need to make the order of evaluation explicit on the syntax level, although not using continuations but *let*-construct. Experimental results would be helpful to have a better idea of the relative costs and powers of these analyses.

Another approach is to allow the programmer to make space reutilization explicit [8,23]. New constructs are added to the language to express sequentiality and destructive updates. A type system, inspired from linear logic, is needed to insure that referential transparency is preserved. Wadler proposed another solution based on monad comprehension [24]; in this case no new typing discipline is necessary. Under this approach the programmer is able and has to reason about order of evaluation and space utilization of the program. This may be regarded as a benefit or as a drawback.

Related are also the work done by Raoult and Sethi [17] who have studied single-threading using a pebble game on a program tree. Let us also mention studies aiming at reducing storage allocations for dynamic data structures [10,12] by doing a sharing analysis on strict, first order, languages.

6.2 Future work

We did not consider product types in this paper. There are several ways to extend this work to deal with such constructs. One straightforward solution is to treat tuples the same way as closures but other approaches should be investigated.

We focused on recursive function parameters because the use of registers seems the most useful in this case. However, Δ may also be used for more general register allocation (e.g. using registers to store intermediate results); Δ can be applied not only to check but also to choose type annotations. For example, let *C* be a continuation expression containing no free variables of types ρ_1 or ρ_2, then when analyzing the expression

$$plusc^{(\alpha \to \kappa) \to \rho 2 \to \rho 1 \to \kappa}\ (plusc^{(\beta \to \kappa) \to \rho 2 \to \alpha \to \kappa}\ C\ y^{\rho 2})\ y^{\rho 2}\ x^{\rho 1}$$

where α and β are type variables, we deduce that the condition for single-threading is $\alpha \not\equiv \rho_2$. Thus $\alpha \equiv \rho_1 \wedge (\beta \equiv \rho_1 \vee \beta \equiv \rho_2)$ are possible choices for implementing those intermediate values by global variables. Predicate Δ would produce a register-interference graph which could be used by classic register allocation algorithms [1].

We believe that the continuation-based approach is also promising for other syntactic analyses. Many properties depend on the evaluation ordering and it is clear that continuations provide valuable information to this respect. We plan to study along these lines stack-single-threading [20] (e.g. detecting when closures or lists can be allocated on a stack) and the introduction of safe destructive updates for list management.

References

1. A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

2. A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *FPCA'89*, pp. 26-38, ACM Press, 1989.

3. M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation order and its application. In *Proc. of 1990 Conf. on Lisp and Func. Prog.,* ACM Press, pp. 242-250, 1990.

4. M. J. Fisher. Lambda-calculus schemata. In *Proc. of the ACM Conf. on Proving Properties about Programs,* Sigplan Notices, Vol. 7(1), pp. 104-109,1972.

5. P. Fradet and D. Le Métayer. Compilation of λ-calculus into functional machine code. In *Proc. TAPSOFT'89*, LNCS vol. 352, pp. 155-166, 1989.

6. P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Trans. on Prog. Lang. and Sys.*, 13(1), pp. 21-51, 1991.

7. C.K. Gomard and P. Sestoft. Globalization and live variables. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* Yale, June 1991. (to appear in Sigplan Notices)

8. J.Guzmán and P. Hudak. Single-threaded polymorphic lambda-calculus. In *IEEE Symposium on Logic in Computer Science,* June 1990.

9. P. Hudak. A semantic model of reference counting and its abstraction. In *Proc. of Conf. on Lisp and Func. Prog.*, ACM Press, pp. 351-363, 1986.

10. K. Inoue, H. Seki and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Trans. on Prog. Lang. and Sys.*, 10(4), 1988, 555-578.

11. N.D. Jones Ed. *Semantics-Directed Compiler Generation*. *LNCS Vol. 94,* 1980.

12. S.B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *FPCA'89,* pp.54-74, ACM Press, 1989.

13. U. Kastens and M. Schmidt. Lifetime analysis for procedure parameters. In *ESOP 86, LNCS Vol. 213,* pp.53-69, 1986.

14. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams. Orbit: An optimizing compiler for Scheme. In *proc. of 1986 ACM SIGPLAN Symp. on Comp. Construction,* 219-233, 1986.

15. D. Lass. Detection of single-threading properties in combinator notations. Ph.D. Thesis, Iowa State University, 1991.

16. G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science 1*, pp. 125-159, 1975.

17. J.-C. Raoult and R. Sethi. The global storage needs of a subcomputation. In *Proc. ACM Symp. on Princ. of Prog. Lang.*, 1984, 148-157.

18. P. Sestoft. Replacing function parameters by global variables. In *FPCA'89,* ACM Press, pp.39-53, 1989. (see also Tech. Report 88-7-2, University of Copenhagen, 1988.)

19. D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Trans. on Prog. Lang. and Sys.*, vol. 7, 1985, 299-310.

20. D.A. Schmidt. Detecting stack-based environments in denotational definitions. *Science of Computer Programming, 11(2),* 1988.

21. D.A. Schmidt. *Denotational Semantics. A Methodology for Language Development.* Allyn & Bacon, 1986.

22. J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass.,1977.

23. P. Wadler. Linear types can change the world! In *IFIP Working Conf. on Programming Concepts and Methods*, North Holland, 1990.

24. P. Wadler. Comprehending monads. In *Proc. of 1990 Conf. on Lisp and Func. Prog., ACM Press*, pp. 61-78, 1990.