

# Polymorphic Time Systems for Estimating Program Complexity

*Vincent Dornic* \*<sup>1,2</sup>

*Pierre Jouvelot* <sup>1,3</sup>

*David K. Gifford* <sup>† 3</sup>

<sup>1</sup>CRI, Ecole des Mines de Paris, France

<sup>2</sup>DRPA, Bull Louveciennes, France

<sup>3</sup>LCS, Massachusetts Institute of Technology, USA

`{dornic,jouvelot}@ensmp.fr`

`gifford@mit.edu`

## Abstract

We present a new approach to static program analysis that permits each expression in a program to be assigned an execution time estimate. Our approach uses a *time system* in conjunction with a conventional type system to compute both the type and the time of an expression. The *time* of an expression is either an integer upper bound on the number of ticks the expression will execute, or the distinguished element `long` that indicates that the expression contains a loop and thus may run for an arbitrary length of time. Every

---

\*Financially supported in Ecole des Mines de Paris by Bull under a CIFRE contract.

†Supported by DARPA/ONR Grant No. N00014-89-J-1988

function type includes a *latent time* that is used to communicate its expected execution time from the point of its definition to the points of its use. Unlike previous approaches a time system works in the presence of first-class functions and separate compilation. In addition, *time polymorphism* allows the time of a function to depend on the times of any functions that it takes as arguments. Time estimates are useful when compiling programs for multiprocessors in order to balance the overhead of initiating a concurrent computation against the expected execution time of the computation. The correctness of our time system is proven with respect to a dynamic semantics.

Categories and Subject Descriptions: D.1.3 [**Programming Techniques**] – Concurrent Programming: *Time system*; D.1.m [**Programming Techniques**] – Miscellaneous: *Type systems*; D.3.1 [**Programming Languages**] – Formal Definitions and Theory; D.3.2 [**Language Classification**] – Applicative Languages; D.3.4 [**Programming Languages**] – Processors: *Compilers, optimization*.

General Terms: Languages, Performance, Verification.

Additional Key Words and Phrases: Time System, Type Systems, Complexity Analysis, Polymorphic Typed Language, Time and Type Checker, Fixpoint Operator.

## 1 Introduction

We present a new approach to static program analysis that permits each expression in a program to be assigned an execution time estimate. Our approach uses a *time system* in conjunction with a conventional type system to compute both the type and the time of an expression. The *time* of an expression is either an integer upper bound on the number of ticks the expression will execute, or the distinguished element `long` that indicates that the expression may contain a loop and thus may run for an arbitrary length of time.

The overall goal of this research is to produce a usable means of estimating at compile

time how long expressions in a higher-order programming language will take to execute. Such information is useful as a documentation for the programmer or as a specification for a library module designer, e.g. for real-time programming. By pointing out the computation kernels (*hot spots*) in large programs, a smart compiler can use it to decide where optimization effort should be concentrated. Ultimately, the most promising application of such time information will be in the determination of useful, as opposed to maximal, parallelism for concurrent evaluation.

As a first step in this direction, we describe below how to distinguish expressions that have a statically bounded execution time from others and, when they are bounded, how to obtain an upper bound approximation of their running time. Although this taxonomy may seem simplistic, previous work suggests that even a simple system for evaluating execution time of expressions can be of great practical value for parallel computing [7,6] or optimizing compilers[2]. For instance, in code generators for parallel MIMD architectures, even a coarse estimate of how long an expression might take to evaluate can be useful when deciding whether a parallelizable construct is worth parallelizing. SIMD compilers need to know whether a function mapped over a vector will take a bounded amount of time to execute since this makes such a mapping expression a good candidate for vector code generation. This information can also be of use in advanced compilers that rely on partial evaluation to perform sophisticated optimizations; knowing whether a given expression can be safely evaluated at compile-time, since it is of bounded time complexity, is of utmost importance. Our system is able to answer such questions

In short, we have developed the first static system for estimating program execution times that permits separate compilation of program components. Our work is an extension of previous work in static semantics, including type and effect systems. Type systems determine what expressions compute, while effect systems determine how expressions compute. These static systems are based upon rules that allow the dynamic properties of programs to be efficiently estimated by a compiler. A static system is said to be *consistent* when it is shown to provide a conservative approximation of the dynamic behavior of expressions.

The basis of our method is a *time system* that annotates function types with their expected *latent time*. Latent times communicate the expected behavior of functions from their point of

definition to their points of use. Execution times are members of a lattice that uses integers to represent precise time estimates and the distinguished element `long` to represent times for functions that may recurse. Time systems are an extension of effect systems [14]. A time system generalizes an effect system by changing its rules to accommodate a lattice of times.

Our work is based upon the assumption that a fully expressive language, supporting first-class functions, is necessary for practical applications and that even very simple time information can be of use for efficient code generation [7]. Unlike other systems for complexity analysis [23,12,18,4,16,8], we do not restrict the source language used by the programmer. Restrictive approaches emphasize computing precise time estimates at the cost of source language expressiveness. Other methods also assume the availability of whole programs in order to assess their time complexity. This is unacceptable when programming *in the large* where modules, such as mathematical libraries, are independently developed and used. Our system alleviates this restriction by extending the type system of the programming language to describe the time complexity of expressions.

In the remainder of this paper, we survey related work (Section 2), describe our source language and its dynamic semantics (Section 3), specify our type and time checking system (Section 4), prove its consistency with the dynamic semantics (Section 5), describe our time estimation algorithm and prove that it is correct (Section 6), show how the fixed point operator  $Y$  fits into our framework (Section 7) and conclude with some observations about possible extensions (Section 8).

## 2 Related Work

Early research showed that it is impossible to statically determine the execution time of arbitrary programs [21]. Subsequent work thus turned to approximations of program execution times. Approximations of program complexity fall into three broad classes called worst-case, best-case and mean-case.

Wegbreit pioneered the field of automatic program complexity analysis with the METRIC

project [23]. His experimental system is able to analyze short and simple Lisp programs. It consists of two separate phases. The first one, called the dynamic phase, extracts a set of mutually recursive equations corresponding to the expressions of the source code. The second one, the static phase, tries to solve this set of equations in order to compute all complexity measures. In this last phase, various methods such as generating function differentiation and matching a data base of known patterns are used. This two-tiered organization is present in almost all of the systems we survey below.

The ACE system [12] uses this framework to analyze FP programs. FP [1] is a purely functional language that manipulates lists; it does not support first-class functions or side effect primitives. The dynamic phase is a rewriting system that extracts and normalizes the recursive equations. A pattern matching process is performed against a data base of standard cases. The results are worst-case growth factors such as “linear” or “polynomial”. Le Metayer does not explain how the ACE data base is maintained.

Sands [18] proposes an extension of Le Metayer’s dynamic phase that handles full functional languages (i.e. with first-class functions) and proves the correctness of his rewriting system. He also presents in [19] an adaptation for lazy high-order languages.

Rosendahl [17] describes and proves correct a rewriting system that, given a Lisp program, returns a *time bound program*, i.e. a program that outputs the computation time instead of its result. The major drawback of his method is that the time bound version of a program may not terminate if the original program does not. It also only accepts programs without first-class functions.

To obtain mean-case results, the distribution of variables’ values over their domains is required. Unfortunately, they are not easy to manipulate and indeed may change during the program execution (e.g. due to side effects in a complex data structure). Flajolet and Vitter bypass this problem [4] by supposing uniform input distributions and analyzing programs that are distribution-transformation free. This class of programs mainly contains tree manipulation algorithms, like tree covering and small rewriting systems (e.g. formal derivation) viewed as tree transformers. A major problem is that the output distribution may not be uniform; this is why

the function composition operator is generally not admissible in their framework.

Ramshaw [16] proposes a set of inference rules that specifies the distribution of variables' values at all program control points. The distribution after a given instruction is computed from the distribution at the previous control point. With this information on distributions, mean-case complexity analysis is possible. Ramshaw's system only deals with first-order control structures (like assignment, test and loops) and simple data structures (such as integers or booleans).

Hickey and Cohen [8] extend Ramshaw's approach to complex data structures by employing Kozen's semantics of stochastic programs [11]. They also propose an extension that covers purely functional languages like FP.

Gray's dissertation addresses the issue of automatic insertion of *futures* in Lisp programs [7] by introducing the notion of *quickness* to model execution time. The quickness attempts to express a non-recursive vs. recursive taxonomy but is closely related to *future* constructs insertion.

Wadler [22] shows that strictness information is useful to compute execution times in lazy languages.

### 3 Language Definition

We present the *CT* language, which will be used throughout the paper. *CT* is a kind of polymorphic strongly typed language [15] in which functions are first-class citizens. It uses a type and time system. *CT* has the full power of the pure lambda calculus. More usual data and control constructs can be trivially desugared in this kernel language; we address the issue of side-effects in the conclusion. Because it is derived *FX-87* language [5], *CT* shares some of its syntax and semantics<sup>1</sup>. The goal of the *FX-87* language design was to accommodate side effects in a functional language by adding effect information into types. We use this approach for *CT* in a time complexity information context.

Our time domain contains the natural integers, which abstract clock ticks, plus a special

---

<sup>1</sup>It is even inspired by *FX-87* in its name since *CT* is to *complexity* what *FX* is to *side-effects*.

value, `long`, describing an unbounded amount of time. Function types in *CT* include a latent time that describes their expected execution time. For example, the classical polymorphic Lisp function `car` will have the *CT* type

```
(poly (t1 type)
      (poly (t2 type)
            (subr 1 ((pairof t1 t2) t1))))
```

where `pairof` is the type constructor for CONS cells (which could be easily added to our base language). The type constructor for abstractions is `subr` and includes the latent time, here `1`, of the function. In the same way, the factorial function type is `(subr long (int) int)`. The type constructor for polymorphic types, `poly`, is used here to abstract the types of the CAR and CDR of the CONS cells.

The *CT* language has three levels of specification, namely the kinds (*k*), the descriptions (*d*, split between types and times) and the expressions (*e*).

$$k \in \text{Kind}$$

$$k ::= \text{type} \mid \text{time}$$

$$d \in \text{Descr}$$

$$d ::= t \mid m$$

$$t \in \text{Type}$$

$$t ::= (\text{drec } i \ t) \quad \text{Recursive Type}$$

$$(\text{subr } m \ (t) \ t) \quad \text{Abstraction Type}$$

$$(\text{poly } (i \ k) \ t) \quad \text{Polymorphic Type}$$

$$i$$

$m \in \text{Time}$	
$m ::= 1 \mid 2 \mid 3 \mid \dots \mid \text{long}$	
$(\text{sumtime } m \ m)$	Time Accumulation
$i$	
$e \in \text{Expr}$	
$e ::= i$	
$(\text{lambda } (i \ t) \ e)$	Abstraction
$(e \ e)$	Application
$(\text{plambda } (i \ k) \ e)$	Polymorphic Abstraction
$(\text{proj } e \ d)$	Projection
$i \in \text{Id}$	Identifier

The expression domain defines the lambda calculus (abstraction and application) with polymorphism (polymorphic abstraction and projection). `lambda` denotes computational abstraction while `plambda` corresponds to type and time abstraction and is a compile-time construct. A `plambda` expression has a `poly` type that describes its polymorphism. The `proj` expression specializes a polymorphic expression to a particular type and can thus be seen as the compile-time equivalent of function application. The following example shows how the polymorphic `car` function whose type is given above can be used, once projected (twice since we only provided here, without loss of generality, single-arity `plambda`):

```
(lambda (p (pairof int int))
  (+ ((proj (proj car int) int) p) 1))
```

This example defines a function that expects a pair `p` of integers and returns its first element incremented by one.

The type of a `lambda` is a `subr` type. A `subr` type includes a latent time that describes the execution time of the function. When function type declarations are introduced by users, they



$$\begin{array}{c}
\text{St}_i \vdash e \rightarrow v, n \\
\hline
\text{St} \vdash (\text{plambda } (i \ k) \ e) \rightarrow v, n \quad [\text{D.Plambda}] \\
\\
\text{St} \vdash e \rightarrow v, n \\
\hline
\text{St} \vdash (\text{proj } e \ d) \rightarrow v, n \quad [\text{D.Proj}]
\end{array}$$

## 4 Type and Time Checking System

The time and type checking system for  $CT$  is composed of two sets of rules. The first one specifies the kind checking of descriptions and the second specifies the type and time checking of expressions. Type and time checking assumes that kind declarations are error-free.

We begin by describing the properties of types and times. The type descriptions admit an equivalence relation  $\sim$ . This relation takes into account the  $\alpha$ -renaming of bound variables in recursive and polymorphic types, but is otherwise structural:

$$\begin{array}{c}
(\text{drec } i \ t) \sim t[i \setminus (\text{drec } i \ t)] \quad [\text{E.Fold}] \\
\\
j \notin \text{FV}(t) \\
\hline
(\text{drec } i \ t) \sim (\text{drec } j \ t[i \setminus j]) \quad [\text{E.Drec}] \\
\\
t_0 \sim t'_0 \\
t_1 \sim t'_1 \\
m \sim m' \\
\hline
(\text{subr } m \ (t_1) \ t_0) \sim (\text{subr } m' \ (t'_1) \ t'_0) \quad [\text{E.Subr}]
\end{array}$$

$$j \notin \text{FV}(t)$$

$$\frac{}{(\text{poly } (i \ k) \ t) \sim (\text{poly } (j \ k) \ t[i \setminus j])} \text{[E.Poly]}$$

where  $d[i \setminus d']$  denotes the substitution of  $i$  by the description  $d'$  in  $d$ .

The binary operator `sumtime` is the additive law of composition on the time domain. Although it is associative and commutative, the time algebra is not a monoid; there is no unit element for the additive law. However, the time `long` is an absorbing element for the `sumtime` operator:

$$\begin{aligned} (\text{sumtime } m_1 \ (\text{sumtime } m_2 \ m_3)) &\sim (\text{sumtime } (\text{sumtime } m_1 \ m_2) \ m_3) \\ (\text{sumtime } m_1 \ m_2) &\sim (\text{sumtime } m_2 \ m_1) \\ (\text{sumtime } m \ \text{long}) &\sim \text{long} \end{aligned}$$

$$(\text{sumtime } m_1 \ m_2) \sim m_1 + m_2 \text{ iff } m_i \neq \text{long}$$

An important aspect of this algebra is shown by the following equation on times:  $m = (\text{sumtime } m \ m_0)$  for which the only solution for  $m$  is `long` for all values of  $m_0$ . This property is important since recursive functions always produce time equations of this form and thus have a `long` latent complexity. We see this admittedly rather limited but nonetheless useful time measure for recursive expressions as a first step towards and a general framework for more specific information about looping constructs, such as *polynomial* or *exponential* complexity in terms of their arguments' size.

The kind rules for types and time follow. TK is a type and kind environment (TK-environment) that maps identifiers to kinds and types. Given a TK-environment, the relation “has kind”, noted  $::$ , maps a description to its kind.

$$\frac{[i :: k] \sqsubseteq \text{TK}}{\text{TK} \vdash i :: k} \text{[K.Env]}$$

$$\begin{array}{c}
\text{TK}[i :: \text{type}] \vdash t :: \text{type} \\
\hline
\text{TK} \vdash (\text{drec } i \ t) :: \text{type} \quad [\text{K.Drec}] \\
\\
\text{TK} \vdash m :: \text{time} \\
\text{TK} \vdash t_i :: \text{type} \\
\hline
\text{TK} \vdash (\text{subr } m \ (t_1) \ t_0) :: \text{type} \quad [\text{K.Subr}] \\
\\
\text{TK}[i :: k] \vdash t :: \text{type} \\
\hline
\text{TK} \vdash (\text{poly } (i \ k) \ t) :: \text{type} \quad [\text{K.Poly}] \\
\\
\text{TK} \vdash m_i :: \text{time} \\
\hline
\text{TK} \vdash (\text{sumtime } m_0 \ m_1) :: \text{time} \quad [\text{K.Sumtime}]
\end{array}$$

The type and time rules for variable, lambda abstraction, application, polymorphic abstraction and projection follow. Added to the classical relation “has type” (noted  $:$ ) is the relation “takes time” (noted  $\$$ ) that denotes an upper bound on the time required to execute an expression.

$$\begin{array}{c}
\text{TK} \vdash e : t \ \$ \ m \\
t \sim t' \ \wedge \ m \sim m' \\
\hline
\text{TK} \vdash e : t' \ \$ \ m' \quad [\text{S.Equiv}] \\
\\
[i : t] \sqsubseteq \text{TK} \\
\hline
\text{TK} \vdash i : t \ \$ \ 1 \quad [\text{S.Env}]
\end{array}$$

$$\begin{array}{c}
\text{TK}[i : t_1] \vdash e : t_0 \ \$ \ m \\
\text{TK} \vdash t_1 :: \text{type} \\
\hline
\text{TK} \vdash (\text{lambda } (i \ t_1) \ e) : (\text{subr } m \ (t_1) \ t_0) \ \$ \ 1 \quad [\text{S.Lambda}] \\
\text{TK} \vdash e_0 : (\text{subr } m_l \ (t_1) \ t_0) \ \$ \ m_0 \\
\text{TK} \vdash e_1 : t_1 \ \$ \ m_1 \\
\hline
\text{TK} \vdash (e_0 \ e_1) : t_0 \ \$ \ (\text{sumtime } (\text{sumtime } m_0 \ m_1) \ (\text{sumtime } m_l \ 1)) \quad [\text{S.Apply}] \\
\text{TK}[i :: k] \vdash e : t \ \$ \ m \\
\hline
\text{TK} \vdash (\text{plambda } (i \ k) \ e) : (\text{poly } (i \ k) \ t) \ \$ \ m \quad [\text{S.Plambda}] \\
\text{TK} \vdash e : (\text{poly } (i \ k) \ t) \ \$ \ m \\
\text{TK} \vdash d :: k \\
\hline
\text{TK} \vdash (\text{proj } e \ d) : t[i \setminus d] \ \$ \ m \quad [\text{S.Proj}]
\end{array}$$

Note that we have not defined any primitive expression for recursion. This is because the `drec` type constructor allows us to express recursive types. Thus the fix point operator `Y` is expressible in our kernel language (see Section 7).

## 5 Consistency

This section states that the static semantics of *CT* is consistent with its dynamic semantics. The proof has two aspects. First, like in any other type checking system, we must prove the consistency between the type of an expression and the value it produces. Note that types contain time information. Then, we must show that the time specified by our static system is an upper bound of the actual execution time as defined in the dynamic semantics.

To accomplish these proofs we will need some additional relations. We will start by introducing a finite map from identifiers to kinds. It corresponds to the “K” of TK.

**Definition 1 (Kind Environment)** *A kind environment, K, is a finite map from identifiers to kinds,  $K \in \text{Id} \rightarrow \text{Kind}$ .*

We can thence define the consistency between values and types, and between states and TK-environments. The function dom returns the domain set of the map given as argument.

**Definition 2** *We define the consistency between a value v and a type t with respect to a kind environment K as the following ternary relation:*

$$\begin{aligned}
K \models v : t &\iff \\
&\text{if } t \sim \text{bool} \text{ then } v \in \text{Bool} \\
&\text{if } t \sim \text{int} \text{ then } v \in \text{Int} \\
&\text{if } t \sim (\text{subr } m (t_1) t_0) \text{ then } v = \langle i, e, \text{St} \rangle \text{ and} \\
&\quad \exists \text{TK s.t. } \left\{ \begin{array}{l} K \models \text{St} : \text{TK} \\ \text{TK}[i : t_1] \vdash e : t_0 \$ m \\ \text{TK} \vdash t_1 :: \text{type} \end{array} \right. \\
&\text{if } t \sim (\text{poly } (i k) t_0) \\
&\quad \text{then } \forall d \in \text{Descr}, K \vdash d :: k \Rightarrow K \models v : t_0[i \setminus d] \\
&\text{if } t \sim (\text{drec } i t_0) \\
&\quad \text{then } K \models v : t_0[i \setminus (\text{drec } i t_0)]
\end{aligned}$$

*In the same way, we define the consistency between a state St and a TK-environment TK with respect to a kind environment K.*

$$\begin{aligned}
K \models \text{St} : \text{TK} &\iff \\
&\left\{ \begin{array}{l} K = \text{TK} \text{ on } \text{Id} \rightarrow \text{Kind} \\ \text{dom}(\text{St}) \subseteq \text{dom}(\text{TK}) \wedge \text{dom}(\text{St}) \cap \text{dom}(K) = \emptyset \\ \forall i \in \text{dom}(\text{St}), K \models \text{St}(i) : \text{TK}(i) \end{array} \right.
\end{aligned}$$

Now, we can express our consistency theorem; note that it only applies to terminating evaluations. It uses the  $\leq$  relation between integers and times which is the natural extension to  $\omega$  of the standard total ordering on integers.

**Theorem 3 (Consistency)** *Let  $St$  be a state,  $e$  an expression,  $v$  a value,  $n$  an integer,  $TK$  a  $TK$ -environment,  $t$  a type,  $m$  a time and  $K$  a kind environment.*

$$\left. \begin{array}{l} St \vdash e \rightarrow v, n \\ TK \vdash e : t \$ m \\ K \models St : TK \end{array} \right\} \implies \left\{ \begin{array}{l} K \models v : t \\ n \leq m \end{array} \right.$$

**Proof (Theorem 3)** By induction on the value of  $n$  and by case analysis of the expressions [3].

□

## 6 Algorithm

Below, we give an algorithm that checks that a given expression is correct with respect to the static semantics. We assume the existence of a kind checking algorithm called  $KCA$  that implements the kind checking rules.

$KCA \in TK\text{-environment} \times Descr \longrightarrow Kind$

$KCA$  is straightforwardly defined by induction on description expressions and is thus omitted.

The algorithm  $TTCA$ :

$TTCA \in TK\text{-environment} \times Expr \longrightarrow Type \times Time$

computes a type and a time description for any expression in a given  $TK$ -environment.

$TTCA(TK, e) = \text{case } e \text{ in}$

$i \quad \Rightarrow \text{if } [i : t] \sqsubseteq TK \text{ then } (t, 1)$

$(\text{lambda } (i \ t_1) \ e)$

$\quad \Rightarrow \text{let } (t_0, m) = TTCA(TK[i : t_1], e)$

```

      ((subr m (t1) t0), 1)
(e0 e1) => let (t, m0) = TTCA(TK, e0)
             let (t1, m1) = TTCA(TK, e1)
             if sim(t, (subr ml (t1) t0))
                then (t0, (sumtime (sumtime m0 m1) (sumtime ml 1)))
(plambda (i k) e)
=> let (t, m) = TTCA(TK[i :: k], e)
    ((poly (i k) t), m)
(proj e d)
=> let (t, m) = TTCA(TK, e)
    let k = KCA(TK, d)
    if sim(t, (poly (i k) t0)) then (t0[i \ d], m)
else => FAIL

```

The pattern-matching function `sim` checks whether two type expressions are similar, i.e. convertible by the  $\sim$  relation. It is defined by structural induction on the type domain with a special proviso for recursive types [9]. The similarity checking of time expressions is slightly more involved since it requires their normalization to either `long` or a lexicographically-sorted list of time variables and a constant.

The following theorem expresses the fact that this algorithm is correct with respect to the static semantics.

**Theorem 4 (Correctness)** *Let  $TK$  be a  $TK$ -environment,  $e$  an expression,  $t_0$  and  $t_1$  two types and  $m_0$  and  $m_1$  two times.*

$$\left. \begin{array}{l} TK \vdash e : t_0 \ \$ \ m_0 \\ (t_1, m_1) = TTCA(TK, e) \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} t_0 \sim t_1 \\ m_0 \sim m_1 \end{array} \right.$$

**Proof** The proof is straightforward and works by structural induction on the expressions and by case analysis on the type and time checking rules □

## 7 An Example: The Fix Point Operator

As discussed above, we can express  $Y$  in terms of the  $CT$  language. When  $Y$  is applied to a potentially recursive function it returns a function of long latent time. When  $Y$  is applied to a non-recursive function it returns a function with the same latent time as its input. Syntactic sugar can be used to package the  $Y$  operator as follows:

$$e ::= \dots$$

$$(\text{rec } (f \ i) \ e)$$

Thus, we could write the factorial function in the following way:

$$\text{FACT} \equiv (\text{rec } (f \ i) \ (\text{if } (= \ i \ 0) \ 1 \ (* \ i \ ((f) \ (- \ i \ 1))))))$$

We use a call-by-value version of the fix point operator  $Y$  because  $CT$  is call-by-value. Looping self-applications in call-by-name  $Y$  are delayed by placing an abstraction around the expression  $(x \ x)$ .

$$Y \equiv \lambda f. (\lambda x. (f \ \lambda. (x \ x)) \ \lambda x. (f \ \lambda. (x \ x)))$$

$$(\text{rec } (f \ i) \ e) \equiv (Y \ \lambda f. \lambda i. e)$$

$Y$  is expressible in our language as follows:

$$Y \equiv$$

$$(\text{plambda } ((t \ \text{type}) (m_1 \ \text{time}) (m_2 \ \text{time}))$$

$$(\text{lambda } ((f \ (\text{subr } m_1$$

$$((\text{subr } (\text{sumtime } m_1 \ 6) \ ()) (\text{subr } m_2 \ (t) \ t)))$$

$$(\text{subr } m_2 \ (t) \ t))))$$

$$((\text{lambda } ((x \ (\text{drec } t_x$$

$$(\text{subr } (\text{sumtime } m_1 \ 3) \ (t_x) \ (\text{subr } m_2 \ (t) \ t))))$$

$$(f \ (\text{lambda } () \ (x \ x))))$$

$$(\text{lambda } ((x \ (\text{drec } t_x$$

```

      (subr (sumtime m1 3) (tx) (subr m2 (t) t))))
(f (lambda () (x x))))))

```

where one can note that  $x$  has a recursive type because it accepts itself as an argument. After self-application, the recursive type disappears. In our definition of  $Y$  we have generalized abstraction and application to multiple arguments.

The type for  $Y$  follows:

```

Y : (poly ((t type)(m1 time)(m2 time))
  (subr (sumtime 6 m1)
    ((subr m1
      ((subr (sumtime 6 m1) () (subr m2 (t) t)))
      (subr m2 (t) t)))
    (subr m2 (t) t)))

```

The type of  $Y$  is used when type and time checking recursive programs, such as the **FACT** example shown above. To obtain the type of **FACT** given below we assumed that the primitives  $=$ ,  $*$  and  $-$  were of latent time 1. Since conditional expressions are viewed as function calls and not special forms in our static semantics, the time of an **if** expression is the **sumtime** of the predicate, consequent, and alternative expressions. A better estimate for **if** that takes into account the non-strictness of conditionals requires a **max** operator in the time algebra.

```

FACT : (poly ((m3 time)(m4 time))
  (subr 1
    ((subr m3 () (subr m4 (int) int)))
    (subr (sumtime 16 m3 m4) (int) int)))

```

Computing the factorial function by applying  $Y$  to **FACT** requires the following projection:

```

((proj Y int 1 long) (proj FACT 7 long)) : (subr long (int) int)

```

When  $Y$  is applied to `FACT`, the projection of  $Y$  is constrained by the following equations. The reader can verify that `long` is the only solution for  $m_4$  in this system:

$$\begin{aligned}
 m_1 &\sim 1 \\
 (\text{sumtime } 6 \ m_1) &\sim m_3 \\
 m_2 &\sim m_4 \\
 t &\sim \text{int} \\
 m_2 &\sim (\text{sumtime } 16 \ m_3 \ m_4)
 \end{aligned}$$

## 8 Conclusion

This paper has introduced the idea of a time system as a new approach for performing time complexity analysis. The key idea in a time system is to use the type of a function to communicate its expected execution time from point of its definition to its point of use.

The time system presented here has two shortcomings. First, it uses a very simple time description domain. Second, it requires programmers to include latent times in their function type declarations. The advantages of our time system include:

- It handles a full functional language with first-class functions. Time polymorphism allows the time of a function to depend on the times of functions it takes as formal parameters.
- It handles separate compilation because latent times communicate time information across compilation boundaries.
- It has been proved consistent with the dynamic semantics.

Extensions of our technique to space analysis and side-effects handling are relatively easy. Static analysis of memory utilization can be accomplished by changing the unit costs used in the time rules to represent storage utilization, and by changing the lambda rule to account for the storage used by closures. Also, by adding a reference type to the set of type constructors, side-effects can be trivially taken into account. The only difficulty lies in the consistency proof which now has to handle a store and use the notion of maximal fixpoint introduced in [20].

We are actively working on extending our system to handle more sophisticated time algebra. For example, we plan to introduce `max` to model the cost of the consequent or the alternative in an `if`, and we plan to do time reconstruction in the vein of [10].

## Acknowledgments

We thank Jean-Pierre Talpin for his help on tricky details of the proof. We also thank Corinne Ancourt and Susan Flynn Hummel for their comments.

## References

- [1] Backus, J. W. Can programming be liberated from Von Neumann style? a functional style and its algebra of programs. *CACM* 21, 8 (August 1978), 613-641.
- [2] Consel, C. Binding time analysis for higher-order untyped functional languages. *Lips and Functional Programming, ACM LFP'90 proceedings, Nice*, (June 1990), 264-273.
- [3] Dornic, V. Analyse de complexité des algorithmes : vérification et inférence. *PhD thesis (in preparation)*, (expected 1992).
- [4] Flajolet, P. and Vitter, J. S. Average-case analysis of algorithms and data structures. *Research report INRIA 718*, (August 1987).
- [5] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *The FX-87 reference manual. Research Report MIT/LCS/TR-407*, (1987).

- [6] Goldberg, F. B. *Multiprocessor execution of functional programs. Research Report YALEU/DCS/RR-618*, (April 1988).
- [7] Gray, S. L. Using futures to exploit parallelism in Lisp. *MIT SB Master Thesis*, (1983).
- [8] Hickey, T. and Cohen, J. Automating program analysis. *JACM* 35, 1 (January 1988), 185-220.
- [9] Jouvelot, P. and Gifford, D. K. The FX-87 Interpreter. *International Conference on Computer Languages, IEEE ICCL'91 proceedings, Miami Beach*, (October 1988).
- [10] Jouvelot, P. and Gifford, D. K. Algebraic reconstruction of types and effects. *Principles on Programming Languages, ACM PoPL'91 proceedings, Orlando*, (January 1991).
- [11] Kozen, D. Semantics of probabilistic programs. *J. Comput. Syst. Sci.* 22 (1981), 328-350.
- [12] Le Métayer, D. ACE: An automatic complexity evaluator. *ACM TOPLAS* 10, 2 (April 1988), 248-266.
- [13] Lucassen, J. M. Types and effects. towards the integration of functional and imperative programming. *PhD dissertation, MIT-LCS*, (September 1987).
- [14] Lucassen, J. M. and Gifford, D. K. Polymorphic effect systems. *Principles on Programming Languages, ACM PoPL'88 proceedings, San Diego*, (January 1988).

- [15] McCracken, N. J. An investigation of a programming language with a polymorphic type structure. *PhD Dissertation, Syracuse University, (1979).*
- [16] Ramshaw, L. H. Formalizing the analysis of algorithms. *Report SL-79-5, Xerox Palo Alto Research Center, Palo Alto, Calif, (1979).*
- [17] Rosendahl, M. Automatic complexity analysis. *Functional Programming Languages and Computer Architecture, ACM FPCA '89 Proceedings, (1989).*
- [18] Sands, D. Complexity analysis for higher order language. *Research Report DOC 88/14, Imperial College, London, (October 1988).*
- [19] Sands, D. Complexity analysis for a lazy high-order languages. *European Symposium On Programming, LNCS ESOP'90 proceedings, (1990).*
- [20] Tofte, M. Operational semantics and polymorphic type inference. *Univ. of Edinburgh, THESIS CST-52-88, (1988).*
- [21] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society, ser 2., vol. 42, 230-265; vol. 43, 544-546, (1936).*
- [22] Wadler, P. Strictness analysis aids time analysis. *Principles on Programming Languages, ACM PoPL'88 proceedings, San Diego, (January 1988).*

[23] Wegbreit, B. Mechanical program analysis. *CACM* 18, 9 (September 1975), 528-539.