Ordering Times

Vincent Dornic
Research Report YALEU/DCS/RR-956
April 1993

This work was supported by the NFS-OCR-9104987

Abstract

We present a type system that provides execution time estimates. Run time estimates are useful for a wide range of applications from automatic parallelization to hot-spot optimizations. We use the framework of effect systems first introduced for the *FX* language. It allows to integrate behavioral information (times) in type descriptions. Our kernel language is powerful enough to model any impure functional language à la *SML*. We prove that our type system is consistent with a standard evaluation scheme.

Résumé

Nous présentons un système de typage orienté vers l'estimation des temps d'exécution. Ces estimations sont utiles pour un large éventail d'applications allant de la parallélisation automatique à l'optimisation des points chauds. Nous utilisons les principes du système d'effets introduit pour le langage *FX*. Ils permettent d'integrer dans les types, des informations décrivant diverses proprietés de l'exécution des programmes. Notre langage est suffisament puissant pour modeliser n'importe quel langage fonctionnel impur comme *SML*. Nous prouvons aussi que notre système de typage est cohérent avec un schema d'évaluation classique.

Contents

1 Introduction

Time estimates are useful for a wide range of applications. For example, in automatic parallelization, the notion of maximal parallelism is usually not sufficient to provide a good speed up as it may generate high overheads for small tasks. Time information is then useful to decide if a program part is large enough to deserve remote execution. In performance debugging of time consuming or real-time applications, you need to locate code hot-spots in programs to perform efficient optimizations. In vectorization, good candidates for vector mapping are functions with a good time invariant property. All theses problems can be solved using execution time information.

It is clear, now, that automatic methods for complexity analysis of programs are strongly needed. Unfortunately, due to the halting problem, statically determining precise execution time is undecidable. To jump over this theoretical barrier, all the previous systems [W75, R79, FV87, L88, S88, HC88] restrict the expressiveness of the programming language. This approach is unacceptable if we want to integrate our method as a module in a compiler.

In [DJG91, D92], we have introduced the notion of *type and time systems*. Time and type systems are an extension of the type and *effect* systems proposed by Lucassen and Gifford in [LG88]. A type describes the value an expression evaluates to while an effect abstracts how the expression compute this value. In [LG88], effects are abstractions of possible side-effects that may happen during execution. These static systems are based on rules that allow behavioral properties of programs to be estimated by the compiler. In short, a time system has two main features: First, a time description abstracting the execution time will be associated to each expression. Second, a *latent time* will be inserted in function types. This latent time communicates the expected behavior of a function its the definition point to its points of application.

The time systems we have proposed are designed for realistic languages. By realistic languages, we mean that their expressiveness are comparable to a commonly used language such as *SML* [HMT89]. We have by-passed the undecidability by computing only a time approximation. A time is then an integer upper bound for the execution time or a special constant, `long`, to denote potentially unbounded computation. Although simple, previous work suggests that even such crude time information can be useful for parallel computing [G83] or optimizing compilers [C90]. We will be more affirmative and claim that, in fact, a precise time description is largely redundant when its information content is used by a compiler. For example, in load-balancing systems, information like *too short* and *long enough* is sufficient and easier to understand (for a compiler) than any time description written using the $O(f(n))$ notation.

In our previous systems, we have encoded recursion with a unique special time constant, `long`. There are two problems with this approach. For example, consider the following piece of program:

```
E ≡ ((lambda (f)
        (rec (g x) E'[f]))
     (rec (g' y) ...  (g' ...)  ...))
```

where `rec` is an operator that allows to define potentially recursive functions. In `(rec (g x) E'[f])`, the expression `E'[f]` is the function body where `x` is the formal parameter and `g` is the internal name of the recursive function. The expression `E` is an application. The parameter part of `E`, is a recursive function definition. It is really recursive as the internal name `g'` is applied to some value in the function body. Then, the defined function has a `long` latent time. Assume that the time of the global expression `E` is `long`. Can we conclude that the formal parameter `f` is really used in `E'[f]`? We cannot because the expression `E'[f]` is itself embedded in a recursive definition. We can not know if the recursion, the time `long`, comes from the internal recursive definition or from the parameter of the application `E`. For the second problem, assume that both recursive definitions lead to real recursive functions. They will both have a `long` latent time. Then, we can not know that the global time (the time of the expression `E`) is in fact the *sum* of two recursions.

The type system we describe in this report is an extension of one presented in [D92]. It allows all basic recursions to be associated with their own time variables. This property solves the two previous problems. If the expression `E` is really composed from two basic recursions, then its time will be the *sum* of two time variables. The time information reconstructed by our time system will be something like:

$$\begin{array}{rll} \texttt{E} & \text{has type} & \texttt{M}_1 \\ \texttt{(lambda (f) (rec (g x) E'[f]))} & \text{has latent time} & \texttt{M}_2 \\ \texttt{(rec (g' y) ... (g' ...) ...)} & \text{has latent time} & \texttt{M}_3 \\ & \textit{with} & \texttt{M}_2 \oplus \texttt{M}_3 \sqsubset \texttt{M}_1 \end{array}$$

where $\texttt{M}_1$, $\texttt{M}_2$ and $\texttt{M}_3$ are time variables subject to the constraint $\texttt{M}_2 \oplus \texttt{M}_3 \sqsubset \texttt{M}_1$ but not further defined. This can be interpreted: `E` *is recursive with time* $\texttt{M}_1$, `(lambda (f) (rec (g x) E'[f]))` *is also recursive with latent time* $\texttt{M}_2$, `(rec (g' y) ... (g' ...) ...)` *is again recursive with latent time* $\texttt{M}_3$ *and* $\texttt{M}_1$ *is strictly greater than* $\texttt{M}_2$ *and* $\texttt{M}_3$.

After an overview of the related work (Section 2), we define (Section 3) the abstract syntax and the dynamic semantics of our language. The evaluation scheme fixed, we describe in details (Section 4) the type and time inference system. We then state and prove the consistency between the time system and the evaluation scheme (Section 5). We show (Section 6) that our type system is more powerful than the previous one. Before concluding, we also show that it enjoys a minimal time property (Section 7).

2 Related Work

Our work is related to two research areas: complexity analysis and type systems.

Complexity Analysis

We are only interested here in the *automatic* complexity analysis of programs. Due to unfortunate and old theoretical restrictions (halting problem [T36]), few automatic systems [W75, L88, S88] [R89, FV87] have been actually proposed. They all manage this inherent difficulty by restricting the expressiveness of the analyzed language or by providing partial information as results. Even in the latter case, no system has been proposed that analyses full-fledged languages with functional and imperative aspects, as this paper presents.

The field of automatic complexity analysis was pioneered by Wegbreit with the METRIC project [W75]. His two-phase system is able to analyze simple first-order *Lisp* programs; the dynamic phase builds from the program a set of mutually recursive equations that model the behavior of the program expressions, which are heuristically solved by the second static phase.

In the Ace system [L88], Le Métayer uses the same framework to analyze first-order FP [B78] programs. The results are worst-case growth factors such as "linear" or "polynomial". The major drawback of this approach is that, beside its restricted input language, it does not suggest how to maintain the correctness of the pattern base used in the static phase. In [S88], Sands proposes a higher-order extension to the previous dynamic phase, also proving its correctness. Wadler [W88] shows that strictness information is useful to compute execution time in lazy languages. Sands integrates this method and proposes in [S90] an automatic system for lazy higher-order languages. Rosendahl [R86, R89] describes an abstract interpretation [AH87] scheme for a pure functional language. Unfortunately, none of these approaches include both first-class functions and imperative constructs.

To obtain mean-case results, the distribution of variables values over their domains is required. Flajolet and Vitter [FV87] assume an uniform input distribution and analyze programs that are distribution-transformation free; this class of programs mainly contains tree manipulation algorithms. A major problem here is that the output distribution may not be uniform, thus making the function composition operator inadmissible. To overcome the distribution problem, Ramshaw [R79] proposes an inference system that specifies the distribution of variables values at all program control points; mean-case complexity analysis is then possible. Hickey and Cohen [HC88] extend Ramshaw's approach to complex data structures by using Kosen's semantics of stochastic programs [K81]. They also propose an extension to purely functional languages like FP.

Type Systems

The two main components of our time system are the effect system framework and the notion of sub-timing. The effect system allows us to introduce behavioral information in type descriptions (execution time estimates in our type system). The sub-timing relation provides more flexibility by coercing expression to greatest execution time estimates.

Effect Systems: The effect system framework was proposed by Lucassen [G86, L87, LG88]. It uses the kinded polymorphic type discipline of [M79] to include into the type of expressions some information about the memory side effects they may perform. To relax the burden on the programmer of having to explicitly provide these effect information, Jouvelot and Gifford [JG91] propose a partial effect reconstruction system. In [TJ91], effect reconstruction is described for a simpler ML-like language but extended to region reconstruction, a region being an approximation of the memory locations in which data structures are allocated. Some of the ideas proposed there will be adopted in the sequel, in particular in the subtiming rule.

In a previous paper [DJG91], we presented the first use of a time system for complexity checking. As in this present paper, the complexity information expresses the possible recursive behavior of expressions. However, instead of simply checking the programmer-provided time information, our new system reconstructs them automatically, provides tighter bounds for test expressions and determines both worst and best case time estimates. The determination of both worst and best case time information allows more program properties to be detected at compile time; for instance, if the best case time of an expression is `long`, then it can be flagged as non-terminating. Finally, our time system has been fully implemented within the μFX-A/DLX compiler [GSTRG92].

Sub-Timing: The sub-timing relation is closely related to the notion of sub-effecting introduced by [TJ91]. Just as sub-typing allows us to coerce an expression to a greater type, sub-effecting can be used to coerce an execution estimate to a greater behavioral description. Hence, the type and effect system specifies an inclusion relation on effects (e.g. times in our system). This inclusion relation is only defined on effects and is not extended to type descriptions.

Our time descriptions are mainly sums of constants (non-zero integers) and order variables. An order variable abstracts a statically unbounded recursion. The sub-timing relation has to be defined with respect to a set of inclusion constraints on order variables. This leads to a type system where all rules are *parametrized* by a constraint set. In [FM88, FM89, FM90], Fuh and Mishra have proposed a general framework for designing type systems allowing such coercions on types. Our time system can be seen as a practical and structural case of this framework. Hence, it enjoys a minimal time property.

3 Language Definition

Our language is a minimal kernel for functional languages (allowing first class functions) with side-effects. Its abstract syntax is represented by the Expr domain. Then, for the functional part, an expression can be an identifier (i), a λ-abstraction, a recursive function (**rec**) or an application. For the non-pure part, we can bind a value to a new location in the store (**new**), we can retrieve a value from the store (**get**) and update a existing location with a new value (**set**). The Id domain stands for identifiers.

$e \in$ Expr		
$e ::= i$		
	(lambda (i) $e)$	Abstraction
	(rec (fi) $e)$	Recursion
	$(e\ e)$	Application
	(new $e)$	ReferenceCreation
	(get $e)$	Dereference
	(set $e\ e)$	Update
$i, f \in$ Id		Identifiers

We now give the evaluation scheme used for our language. This choice is not arbitrary since the way we evaluate an expression can change its execution time. We chosed to use a standard call-by-value evaluation scheme and our time system will be then designed in this way. We define the standard domains for locations (l), basic values (b), values (v), closure, environment (E) and store (st). We add the domain described by n to measure the execution time of an expression. Each basic evaluation operation will be considered as taking one tick to execute.

$l \in$	Loc	Reference
$b \in$	BVal = Bool + Num + {**unit**}	Basic Values
$v \in$	Val = Loc + BVal + Clos	Values
$\langle i, e, E, E\rangle \in$	Clos = Id × Expr × Env × Env	Closures
$E \in$	Env = Id → Val	Environments
$st \in$	Store = Loc → Val	Memories
$n \in$	Num	Positive Integers

As in [TJ91], the *Rec* operator is used in the application rule to unfold the recursive environment (the second one in the closures).

$$\begin{aligned} \text{Rec} &: \text{Env} \to \text{Env} \\ \text{Rec}([\,]) &= [\,] \\ \text{Rec}([f \leftarrow \langle i, e, E', [\,]\rangle]) &= [f \leftarrow \langle i, e, E', [f \leftarrow \langle i, e, E', [\,]\rangle]\rangle] \end{aligned}$$

The Semantics rules are given using judgments of the form:

$$st, E \vdash e \rightarrow v, n, st' \subseteq \text{Store} \times \text{Env} \times \text{Expr} \times \text{Val} \times \text{Num} \times \text{Store}$$

where $st, E \vdash e \rightarrow v, n, st'$ means *With the store st and the environment E, the expression e evaluates to the value v in n ticks, providing a new store st'.*

$$st, E[i \leftarrow v] \vdash i \rightarrow v, 1, st \qquad \text{[D.Env]}$$

$$st, E \vdash (\mathbf{lambda}\ (i)\ e) \rightarrow \langle i, e, E, [\,]\rangle, 1, st \qquad \text{[D.Lambda]}$$

$$st, E \vdash (\mathbf{rec}\ (f\, i)\ e) \rightarrow \langle i, e, E, [f \leftarrow \langle i, e, E, [\,]\rangle]\rangle, 1, st \qquad \text{[D.Rec]}$$

$$\frac{\begin{array}{c} st, E \vdash e_0 \rightarrow \langle i, e, E', E''\rangle, n_0, st_0 \\ st_0, E \vdash e_1 \rightarrow v_1, n_1, st_1 \\ st_1, E' :: \mathrm{Rec}(E'')[i \leftarrow v_1] \vdash e \rightarrow v, n, st' \end{array}}{st, E \vdash (e_0\ e_1) \rightarrow v, 1 + n + n_0 + n_1, st'} \text{[D.Apply]}$$

$$\frac{\begin{array}{c} st, E \vdash e \rightarrow v, n, st_0 \\ [l \leftarrow v'] \not\subseteq st_0 \end{array}}{st, E \vdash (\mathbf{new}\ e) \rightarrow l, n+1, st_0[l \leftarrow v]} \text{[D.New]}$$

$$\frac{\begin{array}{c} st, E \vdash e \rightarrow l, n, st_0 \\ [l \leftarrow v] \subseteq st_0 \end{array}}{st, E \vdash (\mathbf{get}\ e) \rightarrow v, n+1, st_0} \text{[D.Get]}$$

$$\frac{\begin{array}{c} st, E \vdash e_0 \rightarrow l, n_0, st_0 \\ st_0, E \vdash e_1 \rightarrow v, n_1, st_1 \end{array}}{st, E \vdash (\mathbf{set}\ e_0\ e_1) \rightarrow \mathbf{unit}, n_0 + n_1 + 1, st_1[l \leftarrow v]} \text{[D.Set]}$$

There is only one rule for each kind of expression. The evaluator is then deterministic and syntax-directed. An identifier (axiom D.Env) evaluates to the value it is bound to in the environment E. The store is not modified and the time needed for the evaluation is one tick. A λ-abstraction (axiom D.Lambda) evaluates to a non recursive closure (the second environment is empty) in one tick. On the other hand, a recursive function definition (axiom D.Rec) evaluates to a recursive closure. The second environment contains a binding from f to the non recursive version of the function. At each application (rule D.Apply), this second environment is unfolded one step to provide recursion. The (non-commutative) operator :: denotes for environment concatenation. Remaining rules deal with side-effects. Those rules define a perfectly safe evaluation scheme as we always allocate values to unbound locations and only access or update bound locations.

Note that our language contains no test expression. In fact, it is sufficiently powerful to express one by an abstraction embedding.

4 Type and Time Inference System

We now give the specification of the time system. Our language being implicitly typed, the whole types and times descriptions have to be reconstructed. But in fact, we are only interested in time information. We have then reduced type descriptions to the minimum necessary to perform time reconstruction.

The specification consists of some domain definitions, a set of equations describing time equivalences, a set of rules specifying the notion of *time inclusion*, and the inference rules showing how to associate times to expressions.

$$\begin{array}{l} \delta \in \text{Descr} \\ \delta ::= t \mid m \end{array}$$

$$\begin{array}{l} m \in \text{Time} \\ m ::= n \mid M \mid m \oplus m \end{array}$$

$$\begin{array}{l} n \in \text{Num} \\ n ::= \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \ldots \end{array}$$

$$\begin{array}{l} M \in \text{Order} \\ M ::= M_1 \mid M_2 \mid M_3 \mid \ldots \end{array}$$

$$\begin{array}{l} t \in \text{Type} \\ t ::= \mathtt{B} \mid \mathtt{ref}\ t \mid t \xrightarrow{m} t \end{array}$$

$$\mathrm{T} \in \text{TEnv} = \text{Id} \rightarrow \text{Type}$$

$$\Delta \in \text{CSet} = \mathcal{P}(\text{Order} \times \text{Order})$$

A description (δ) is a type or a time. A time (m) can be (n) a integer binding the execution time, an order (M) abstracting a recursion or the sum ($\oplus$) of two (or more) times. An integer is always different from 0. The set Order is a set of variables. A type (t) can be a basic type (B), a reference type $\mathtt{ref}\ t$ or an arrow type $t \xrightarrow{m} t'$ for functions of domain t, codomain t' and latent time m. A type environment (T) is a map from identifiers to types. A constraint set (Δ) is a set of order couples. Constraints sets will be used to generate the time inclusion relation.

$$\begin{array}{ll} m_1 \oplus (m_2 \oplus m_3) \sim (m_1 \oplus m_2) \oplus m_3 & \text{Associativity} \\ m_1 \oplus m_2 \sim m_2 \oplus m_1 & \text{Commutativity} \\ \\ n_1 \oplus n_2 \sim n_1 + n_2 & \text{Additivity} \\ \\ M \oplus n \sim M & \text{Absorbency} \\ M \oplus M \sim M & \text{Idempotency} \end{array}$$

We can see the time set with the sum operator (Time, $\oplus$) as an algebra on times. The equivalence between descriptions is denoted $\sim$. The $\oplus$ operator is associative and commutative. The sum of two (or more) bounded times can be reduced by adding the integers in the standard way. Orders represent unbounded computation and are considered as always greater than any bounded time. The last equation says that twice the *same* recursion is, in terms of execution time, equivalent to one recursion. Remember that we want only to detect each basic recursion and we are not interested in knowing how many times they really happen.

As we have already mentioned, constraint sets generate a partial order on time descriptions. This *time inclusion* is defined by the relation:

$$\Delta \vdash m \sqsubset m' \subseteq \text{CSet} \times \text{Time} \times \text{Time}$$

which means that the time m is less than the time m' with respect to the constraint set Δ. Note that there is not only one inclusion relation on times but a whole set generated by all possible constraint sets. We will see below how constraint sets are defined. For instance, consider a constraint set as a kind of oracle.

$$\frac{n \leq n'}{\Delta \vdash n \sqsubset n'}\ [\text{I.Num}]$$

$$\Delta \vdash n \sqsubset M \qquad [\text{I.Mix}]$$

$$\frac{\{M \sqsubset M'\} \subseteq \Delta}{\Delta \vdash M \sqsubset M'}\ [\text{I.Order}]$$

$$\Delta \vdash m \sqsubset m \qquad [\text{I.Reflex}]$$

$$\frac{\Delta \vdash m_0 \sqsubset m_1 \quad \Delta \vdash m_1 \sqsubset m_2}{\Delta \vdash m_0 \sqsubset m_2}\ [\text{I.Trans}]$$

$$\frac{\Delta \vdash m_0 \sqsubset m_1 \quad m_0 \sim m_0' \quad m_1 \sim m_1'}{\Delta \vdash m_0' \sqsubset m_1'}\ [\text{I.Equiv}]$$

$$\frac{\Delta \vdash m_0 \sqsubset m_1 \quad \Delta \vdash m_2 \sqsubset m_3}{\Delta \vdash m_0 \oplus m_2 \sqsubset m_1 \oplus m_3}\ [\text{I.Sum}]$$

When time descriptions are bounded times (rule I.Num), the standard order $\leq$ on integers holds. A bounded time is always less than an order (rule I.Mix). Order inclusion can be extracted from constraint sets (rule I.Order). The two following rules (I.Reflex and I.Trans) define the standard laws of partial orders. The time inclusion commutes with time equivalence (rule I.Equiv) and time sum (rule I.Sum). Note that the only condition needed on Δ for the inclusion relation to be a partial order is that Δ contains no cycles. Otherwise, Δ is free and, for example, can be empty.

The usefulness of an inclusion relation like this can be illustrated by the following:

$$\Delta \vdash m \oplus \mu \sqsubset \mu$$

where μ is an unknown time variable. Assume, we want to solve this time inclusion, i.e. to find the least solution for μ. Whatever the time m is, we know that μ can not be a integer. Lets say that μ is an order M. The time inclusion is then valid if we can prove it using the inclusion rules. If the time m is an integer, then this inclusion is always verified. If m is the sum of some orders, then we have to deduce (with the inclusion rules) that M is greater than all thoses orders. This will be possible if for each M' in the sum m we can prove that $\Delta \vdash M' \sqsubset M$. All thoses order inclusions can be obtained by only two rules: I.Order or I.Trans. From the rule I.Order, we know that the constraint $\{M' \sqsubset \emptyset\}$ is in Δ. From the rule I.Trans, we know that it exists an order variable *between* M' and M, i.e. an order M'' such that $\Delta \vdash M' \sqsubset M''$ and $\Delta \vdash M'' \sqsubset M$. As theses two new inclusions are obtained by the sames rules, we can conclude that Δ has to contain a minimal set of constraints powerful enough to prove all the requested inclusions.

This property is the keystone of our type system since all recursive function definitions lead to time inclusions of this form. hence, for a recursive definition to be type (and time) correct, the constraint set Δ has to contain at least the constraints needed to deduce all time inclusions

generated by the recursive definitions. As each recursive function is associated with its own order variable, we will be able, by examining the constraint set Δ, to say how the various basic recursions of the whole program are related.

$$\mathtt{B} \sim \mathtt{B} \qquad \text{[E.Basic]}$$

$$\frac{t_0 \sim t_0' \quad t_1 \sim t_1' \quad m \sim m'}{t_1 \xrightarrow{m} t_0 \sim t_1' \xrightarrow{m'} t_0'} \text{[E.Subr]}$$

$$\frac{t \sim t'}{\mathtt{ref}\ t \sim \mathtt{ref}\ t'} \text{[E.Ref]}$$

The equivalence relation on type descriptions is straightforward. Arrow types are equivalents (rule E.Subr) if the domains, the codomains and the latent times are respectively equivalents. Reference types are equivalent (rule E.Ref) if referenced types are equivalent.

The time system follows. There is one rule by expression constructor and then the time system is deterministic. Rules specify how to associate time (and type) to expressions and use judgments like:

$$\Delta, \mathrm{T} \vdash e : t \$ m \subseteq \mathrm{CSet} \times \mathrm{TEnv} \times \mathrm{Expr} \times \mathrm{Type} \times \mathrm{Time}$$

which mean: *With the constraint set Δ and the type environment* T, *the expression e has type t and time m.*

An identifier i (rule S.Env) has type t and time 1 if it is bound to t in T. We assume that the time to access a variable is constant. This is usually the case in efficient implementations where the symbol table is, for example, a hash table. In the rule S.Lambda, the time m of the function body becomes the latent type of the function. The same principle is used in the rule S.Rec but the time of the function body m_0 can be a sum containing itself (as it is also the latent time of the internal recursive function $\mathtt{f}$). Then, m_0 will have to satisfy a recursive equation as discussed before and will lead to a new order variable. The latent time of a function is extracted when applied (rule S.Apply) to a parameter. The rules for the side-effect constructors (S.New, S.Get and S.Set) are all straightforward and just add one execution step to times. Finally, rule S.Takes allows the time associated with an expression to be longer value. This rule is especially useful in recursive typing to equate the latent time of the internal recursive function and the time of the function body. It allows also to get more flexibility in application by coercing the time of the parameter to a higher one.

$$\Delta, \mathrm{T}[i : t] \vdash i : t \$ 1 \qquad \text{[S.Env]}$$

$$\frac{\Delta, \mathrm{T}[i : t_1] \vdash e : t_0 \$ m}{\Delta, \mathrm{T} \vdash (\mathtt{lambda}\ (i)\ e) : t_1 \xrightarrow{m} t_0 \$ 1} \text{[S.Lambda]}$$

$$\frac{\Delta, \mathrm{T}[f : t_1 \xrightarrow{m_0} t_0, i : t_1] \vdash e : t_0 \$ m_0}{\Delta, \mathrm{T} \vdash (\mathtt{rec}\ (f i)\ e) : t_1 \xrightarrow{m_0} t_0 \$ 1} \text{[S.Rec]}$$

$$\frac{\Delta, \mathrm{T} \vdash e_0 : t_1 \xrightarrow{m} t_0 \,\$\, m_0 \quad \Delta, \mathrm{T} \vdash e_1 : t_1 \,\$\, m_1}{\Delta, \mathrm{T} \vdash (e_0\ e_1) : t_0 \,\$\, m_0 \oplus m_1 \oplus m \oplus \mathbf{1}}\ [\text{S.Apply}]$$

$$\frac{\Delta, \mathrm{T} \vdash e : t \,\$\, m}{\Delta, \mathrm{T} \vdash (\mathbf{new}\ e) : \mathbf{ref}\ t \,\$\, m \oplus \mathbf{1}}\ [\text{S.New}]$$

$$\frac{\Delta, \mathrm{T} \vdash e : \mathbf{ref}\ t \,\$\, m}{\Delta, \mathrm{T} \vdash (\mathbf{get}\ e) : t \,\$\, m \oplus \mathbf{1}}\ [\text{S.Get}]$$

$$\frac{\Delta, \mathrm{T} \vdash e_0 : \mathbf{ref}\ t_1 \,\$\, m_1 \quad \Delta, \mathrm{T} \vdash e_1 : t_1 \,\$\, m_0}{\Delta, \mathrm{T} \vdash (\mathbf{set}\ e_0\ e_1) : \mathbf{B} \,\$\, m_0 \oplus m_1 \oplus \mathbf{1}}\ [\text{S.Set}]$$

$$\frac{\Delta, \mathrm{T} \vdash e : t \,\$\, m \quad \Delta \vdash m \sqsubset m'}{\Delta, \mathrm{T} \vdash e : t \,\$\, m'}\ [\text{S.Takes}]$$

As we said before, a constraint set must not contain cycles in order to give rise to a partial order on times. This is expressed by the following definition:

Definition 1 *A constraint set Δ is consistent if the following implication holds for all order variables M and M':*

$$\Delta \vdash M \sqsubset M' \implies \{M' \sqsubset M\} \not\subseteq \Delta$$

We can now define what is a correct typing:

Definition 2 (Type Correctness) *An expression e is type correct if there is a consistent constraint set Δ, a type environment T, a type t and a time m such that $\Delta, \mathrm{T} \vdash e : t \,\$\, m$. We say that the judgment $\Delta, \mathrm{T} \vdash e : t \,\$\, m$ holds or is valid.*

It may seem strange that we do not further define the constraint set Δ. In fact, we will see that, if an expression is type correct, then there are many possible choices for Δ. Fortunately, the type system enjoys a minimal time property and hence there is a minimal correct typing such that all other correct typings are just instances of the minimal one.

Together with this minimal time property, we have also to prove that the static and the dynamic semantics are consistent.

5 Consistency

To show the consistency between the static and dynamic semantics, we must prove two facts. First fact, for all expressions, the value to which it evaluates is consistent with its type. Second fact, for all expressions, the time the expression needs to evaluates is consistent with the expression time. As we will see, the two facts cannot be proved only by restating the main theorem of [D92]. This theorem (SDC) only deals with type/value and execution time/bounded time consistency and says nothing about the detection of each basic recursion.

5.1 Restating the SDC Theorem

The complete proof is similar to the one presented in detail in [D92]. We only give the principal definitions and lemmas.

Our language possesses side-effect primitives. Proving the consistency between a value and a type can depend upon some values in the store. To express the type tagging of the values in the store, we start with two definitions:

Definition 3 (Abstract Store) *An abstract store is finite map from locations to types.*

$$ST \in \text{AbStore} = \text{Loc} \rightarrow \text{Type}$$

Definition 4 (Typed Store) *A typed store is a tuple made from a store st, an abstract store ST and a constraint set* Δ *such that* $\text{dom}(st) = \text{dom}(ST)$. *We write* $st : ST, \Delta$.

The next definition deals with the extension of the store with new bindings. It is the straightforward extension of the inclusion on functions to couples of functions.

Definition 5 (Inclusion) *A typed store* $st : ST, \Delta$ *is included in another one* $st' : ST', \Delta'$, *noted* $st : ST, \Delta \subseteq st' : ST', \Delta'$, iff $st \subseteq st'$, $ST \subseteq ST'$ *and* $\Delta \subseteq \Delta'$.

The following property specifies what means for a value to be consistent with a type. Note, that it depends upon a typed store and a constraint set.

Property 6 ($\models :$)

$$\begin{aligned}
st : ST, \Delta \models v : t \iff\ & \text{if } v = b \text{ then } t \sim \mathtt{B} \\
& \text{if } v = \langle i, e, E, [\,] \rangle \\
& \quad \text{then } \exists t_1, t_0, m, \mathrm{T} \text{ s.t. } t \sim t_1 \xrightarrow{m} t_0 \wedge \\
& \qquad st : ST, \Delta \models E : \mathrm{T} \wedge \Delta,\ \mathrm{T}[i : t_1] \vdash e : t_0\ \$\ m \\
& \text{if } v = \langle i, e, E, [f \leftarrow \langle i, e, E, [\,] \rangle] \rangle \\
& \quad \text{then } \exists t_1, t_0, m, \mathrm{T} \text{ s.t. } t \sim t_1 \xrightarrow{m} t_0 \wedge \\
& \qquad st : ST, \Delta \models E : \mathrm{T} \wedge \Delta,\ \mathrm{T}[f : t_1 \xrightarrow{m} t_0, i : t_1] \vdash e : t_0\ \$\ m \\
& \text{if } v = l \\
& \quad \text{then } \exists v', t' \text{ s.t. } t \sim \mathtt{ref}\ t' \wedge \\
& \qquad [l \leftarrow v'] : [l : t'],\ \Delta \subseteq st : ST, \Delta \wedge st : ST, \Delta \models v' : t'
\end{aligned}$$

$$st : ST, \Delta \models E : \mathrm{T} \iff \text{dom}(E) = \text{dom}(\mathrm{T}) \wedge \forall i \in \text{dom}(E),\ st : ST, \Delta \models E(i) : \mathrm{T}(i)$$

It remains to say what means for an execution time to be consistent with a time. Note that the following property says nothing about the detection of basic recursions and treats every recursion as being more expensive than any finite computation.

Property 7 ($\models \leq$)

$$\begin{aligned}
\models n \leq m \iff\ & \text{if } m \in \text{Order then } \mathit{true} \\
& \text{if } m \notin \text{Order then } n \leq m
\end{aligned}$$

The last definition expresses the effects of a computation on the store. A store will *succeed* to another if it can be obtained by any sequence of allocations and type preserving updates on the beginning store.

Definition 8 (Succession) *A typed store* $st' : ST', \Delta$, *succeeds to another* $st : ST, \Delta$, *noted* $st : ST, \Delta \sqsubseteq st' : ST', \Delta'$, *iff*

$$\begin{cases} ST \subseteq ST' \\ \forall v, t, st : ST, \Delta \models v : t \Rightarrow st' : ST', \Delta \models v : t \end{cases}$$

We can now restate the static and dynamic consistency theorem (SDC). It says that *if an expression e has type t and time m, if it evaluates to a value v in n ticks and if the beginning environments are consistents, then the computed value is consistent with the associated type, the number of ticks is consistent with the associated time and the store after the evaluation succeed to the beginning one.*

Theorem 9 (SDC) *Let st and st' be stores, ST an abstract store,* Δ *a constraint set, e an expression,* T *a type environment, t a type, m a time, E an environment, v a value and n an integer, then the following implication holds:*

$$\left.\begin{array}{l} \Delta, \mathrm{T} \vdash e : t \$ m \\ st, E \vdash e \rightarrow v, n, st' \\ st : ST, \Delta \models E : \mathrm{T} \end{array}\right\} \Longrightarrow \exists ST' \text{ and } \Delta' \text{s.t.} \left\{\begin{array}{l} st : ST, \Delta \sqsubseteq st' : ST', \Delta' \\ st' : ST', \Delta' \models v : t \\ \models n \leq m \end{array}\right.$$

Proof (SDC) It is similar to the proof presented in [D92]. We just have to propagate the constraint set Δ everywhere. The monotonicity of the $\mathcal{F}$ operator is easy to prove. Then the maximum fixpoint induction can be used to prove the two main lemmas (expansion and modification). The proof of the theorem itself works in exactly the same way, i.e. by induction on the execution time (n) and by case analysis on the expressions. □

5.2 Basic Recursions

The consistency between execution times and bounded times is established by the previous theorem. We now focus on recursions and show that the execution time of an expression is mainly the *sum* of its basic recursions. It may seem necessary to use the notion of evaluation explicitly, as in the previous theorem, to say that the remaining execution time, when all basic recursions are *deleted*, is bounded by some constant. In fact, we just have to show that this remaining time is a bound time and then, by the previous theorem, we know that its execution time is bounded.

We show that, when all sub-expressions providing basic recursions to a global expression are substituted by expressions of bounded time, the global expression has a bounded time. We start by defining the set of potential recursive sub-expressions of an expression, i.e. the set of sub-expressions that may create recursion.

Definition 10 (Recursive Sub-Expressions) *let* rse(E) *be the set of potentially recursive sub-expression of the expression E. By potentially recursive, we mean that they are of the form* (**rec** (fi) e).

Note that rse(E) is a set of recursive definitions and that these sub-expressions are always of time 1 in the time system. In fact, these expressions are function definitions that have unbounded latent times.

The question is now: When an expression is recursive, where does this recursion come from? The following lemma states that there is always a recursive sub-expression providing it.

Lemma 11 *Let* Δ *be a constraint set,* T *an environment, E an expression, t a type, m a time and M an order. Then, the following implication holds:*

$$\Delta, \mathrm{T} \vdash E : t \$ m \oplus M \Longrightarrow \exists t_1, t_0, m' \text{ and } e \in \mathrm{rse}(E) \text{ s.t. } \Delta, \mathrm{T} \vdash e : t_1 \xrightarrow{M \oplus m'} t_0 \$ 1$$

Proof By disjunction and induction on the expressions. At first, we see that E can not be a identifier (i), and abstraction ((**lambda** (i) e)) nor a recursive abstraction ((**rec** (fi) e)). The other cases have each to be checked.

Case $\boxed{E = (e_0\ e_1)}$
We know by the S.Apply rule that $\Delta, t \vdash E : t_0\ \$\ m_0 \oplus m_1 \oplus m_l \oplus \mathbf{1}$. This implies the equivalence $M \oplus m \sim m_0 \oplus m_1 \oplus m_l \oplus \mathbf{1}$. If M is in m_l then the implication holds. Otherwise, the order M is in the time m_0 (respectively m_1), then the implication holds by induction on the expression e_0 (respectively e_1).

Case $\boxed{E = (\mathbf{new}\ e_0)}$
We know by the rule S.New that $\Delta, \mathrm{T} \vdash E : \mathbf{ref}\ t_0\ \$\ m_0 \oplus \mathbf{1}$. Then, we have $M \oplus m \sim m_0 \oplus \mathbf{1}$ and by absorbency $M \oplus m \sim m_0$. The implication holds by induction on e_0.

Case $\boxed{E = (\mathbf{get}\ e_0)}$
Similar to the previous one.

Case $\boxed{E = (\mathbf{set}\ e_0\ e_1)}$
Similar to the application without the case of the latent time. □

No all recursive sub-expressions participate to the global time. For example, in the following program:

```
E ≡ ((lambda (f) 1)
     (rec (g y) ...  (g ...)  ...))
```

The application parameter is a recursive sub-expression but, as it is never applied, its latent time will never appear the time of the expression E.

Definition 12 (Significant Sub-Expressions) *The set of significant sub-expressions in the expression E with the constraint set Δ and the environment* T *is:*

$$\mathrm{sse}_\Delta(\mathrm{T},E) = \{e \mid \Delta, \mathrm{T} \vdash E : t\ \$\ m \oplus M,\ \Delta, \mathrm{T} \vdash e : t_1 \xrightarrow{M \oplus m'} t_0\ \$\ \mathbf{1} \text{ and } e \in \mathrm{rse}(E)\}$$

Remember that our language allows side-effects and first-order functions. A sub-expression could be recursive and not significant in a more complicated way. For example, a powerful data constructor (e.g. the **defstruct** of *Common Lisp*) could returns lots functions for data manipulations some of which might be recursive sub-expressions.

The following lemma states that if an expression has a bounded time, then it contains no significant sub-expressions.

Lemma 13 *If* $\Delta, \mathrm{T} \vdash E : t\ \$\ n$ *then* $\mathrm{sse}_\Delta(\mathrm{T},E) = \emptyset$.

Proof Straightforward. □

We know define how to *delete* significant sub-expressions. We start by substituting one of them by a non-significant one. All sub-expression are considered tagged with their unique path in the structural tree. All equivalence relations (e.g. through α-renaming) are ignored and all sub-expressions are considered to be distinct.

Definition 14 (Reduced Expression) *We define the reduction of the expression E by a sub-expression e in the constraint set Δ and the environement* T *as:*

$$\mathrm{red}^e_\Delta(\mathrm{T},E) = (\mathrm{T}[i : t_1 \xrightarrow{\mathbf{1}} t_0], E[e \backslash i])$$

where $\Delta, \mathrm{T} \vdash e : t_1 \xrightarrow{m} t_0\ \$\ \mathbf{1}$ *and* $i \notin \mathrm{FV}(E)$.

When an expression E is reduced by a sub-expression e, e is no longer a significant sub-expression.

Lemma 15 *Let E be an expression, e a sub-expression, Δ a constraint set and T an environment, then the following implication holds:*

$$e \in \mathrm{sse}_\Delta(\mathrm{T},E) \Longrightarrow e \notin \mathrm{sse}_\Delta(\mathrm{red}^e_\Delta(\mathrm{T},E))$$

Proof In fact, e is no longer in E as it has been substituted. □

We now *delete* all significant sub-expressions. The reduced expression is *minimal* when all its significant sub-expressions have been substituted by non-significant ones.

Definition 16 (Minimal Reduced Expression) *Let $\mathrm{sse}_\Delta(\mathrm{T},E)$, the set of sub-expressions of E, be equal to $\{e_0, ..., e_n\}$. We define the maximal reduced expression of E in Δ and T as:*

$$\mathrm{mred}_\Delta(\mathrm{T},E) = \mathrm{red}^{e_0}_\Delta(... \mathrm{red}^{e_n}_\Delta(\mathrm{T},E) ...)$$

Lemma 17 *For any expression E, environment T and constraint set Δ, $\mathrm{sse}_\Delta(\mathrm{mred}_\Delta(\mathrm{T},E)) = \emptyset$*

A minimal reduced expression has a bounded time.

Theorem 18 *Let E be an expression, Δ a constraint set and T an environment. Assume that $(\mathrm{T}',E') = \mathrm{mred}_\Delta(\mathrm{T},E)$ then there is a type t' and a time n such that $\Delta, \mathrm{T}' \vdash E' : t' \$ n'$.*

Proof Assume that $(\mathrm{T}',E') = \mathrm{mred}_\Delta(\mathrm{T},E)$. Assume also there exist a type t' and a time m such that $m \not\sim n$. As m is not a bounded time, it is a sum of orders. So, there is a time m' and an order M such that $m \sim m' \oplus M$. By the lemma 11, there are t_1, t_0, m' and $e \in \mathrm{rse}((\mathrm{mred}_\Delta(\mathrm{T},E)))$ such that $\Delta, \mathrm{T}' \vdash e : t_1 \xrightarrow{m' \oplus M} t_0 \$ 1$. But, in this case, the expression e is in $\mathrm{sse}_\Delta(\mathrm{mred}_\Delta(\mathrm{T},E))$ and we know that $\mathrm{sse}_\Delta(\mathrm{mred}_\Delta(\mathrm{T},E)) = \emptyset$. By contradiction, we get $m \sim n$. □

Two questions remain. First, are all significant sub-expressions really significant in practice? In other words, is the set $\mathrm{sse}_\Delta(\mathrm{T},e)$ minimal with respect to the evaluation scheme? The answer is: no. Consider the following example:

```
E ≡ ((lambda (f)
       (if true
           1
         (f ...)))
     (rec (g y) (...  (g ...)  ...)))
```

The application parameter is in the set $\mathrm{sse}_\Delta(\mathrm{T},e)$, but is never applied in practice. Fortunately, this kind problem is not linked to our time system but appears in all type systems. We can not expect to solve it in any way.

Second, is this time system more powerful then the previous one presented in [D92]? The answer, as we will see in the next section, is: yes.

6 Comparison

In the [D92] time system, all the recursive sub-expressions have a **long** latent time (i.e. not bounded). It is then unable to detect what sub-expression are significant.

Definition 19 *In this section, we use $\mathrm{T} \vdash_\omega e : t \$ m$ when refering to the time system of [D92].*

We show that it can be obtained as a particular instance of the time system presented in this report by setting Δ to be the empty set and Order to be the singleton {**long**}.

Theorem 20 *Let* T *be an environment, e an expression, t a type and m a time. If the set* Order *is equal to the sigleton* {**long**} *then the following equivalence holds:*

$$\emptyset, \mathrm{T} \vdash e : t \$ m \Longleftrightarrow \mathrm{T} \vdash_\omega e : t \$ m$$

Proof It is straightforward to show equivalence between each couple of static rules using the restriction defined by Order = {**long**} and $\Delta = \emptyset$. We must also prove that the inclusion with respect to Δ (i.e. $\Delta \vdash m \sqsubset m'$) reduces to the previous time inclusion (without constraint set). This point is obvious. □

To compare the expressiveness of two time systems, we need to define what are the set of significant sub-expressions ($\mathrm{sse}_\omega(\mathrm{T},E)$), the one-step reduction of an expression ($\mathrm{red}^e_\omega(\mathrm{T},E)$) and the minimal reduction ($\mathrm{mred}_\omega(\mathrm{T},E)$) with respect to the typing $\mathrm{T} \vdash_\omega e : t \$ m$.

Definition 21 *We define* $\mathrm{sse}_\omega(\mathrm{T},E)$, $\mathrm{red}^e_\omega(\mathrm{T},E)$ *and* $\mathrm{mred}_\omega(\mathrm{T},E)$ *as the reductions of the previous defined ones:*

$$\begin{aligned}
\mathrm{sse}_\omega(\mathrm{T},E) &= \{e \mid \mathrm{T} \vdash_\omega E : t \$ \mathbf{long},\ \mathrm{T} \vdash_\omega e : t_1 \xrightarrow{\mathbf{long}} t_0 \$ 1 \text{ and } e \in \mathrm{rse}(E)\} \\
\mathrm{red}^e_\omega(\mathrm{T},E) &= (\mathrm{T}[i : t_1 \xrightarrow{1} t_0], E[e \backslash i]) \\
&\quad \textit{where } \mathrm{T} \vdash_\omega e : t_1 \xrightarrow{\mathbf{long}} t_0 \$ 1 \textit{ and } i \notin \mathrm{FV}(E) \\
\mathrm{mred}_\omega(\mathrm{T},E) &= \mathrm{red}^{e_0}_\omega(\ldots \mathrm{red}^{e_n}_\omega(\mathrm{T},E) \ldots)
\end{aligned}$$

The new time system is at least as powerful as the old one.

Lemma 22 *Let* Δ *be a constraint set,* T *a type environment and E an expression then the following inclusions hold:*

$$\mathrm{sse}_\Delta(\mathrm{T},E) \subseteq \mathrm{sse}_\omega(\mathrm{T},E) \subseteq \mathrm{rse}(E)$$

Proof First, the inclusion in $\mathrm{rse}(E)$ is straightforward, coming from the definition of $\mathrm{sse}_\omega(\mathrm{T}, E)$. For the other inclusion, assume $e \in \mathrm{sse}_\Delta(\mathrm{T},E)$ then, by definition, we know that $\Delta, \mathrm{T} \vdash E : t \$ m \oplus M$ and $\Delta, \mathrm{T} \vdash e : t_1 \xrightarrow{M \oplus m'} t_0 \$ 1$. Now, by reducing (i.e. stating $\Delta = \emptyset$ and Order = {**long**}) we obtain $\mathrm{T} \vdash_\omega E : t \$ \mathbf{long}$ and $\mathrm{T} \vdash_\omega e : t_1 \xrightarrow{\mathbf{long}} t_0 \$ 1$. □

But, the old one is less powerful than the new one.

Property 23 *Let* Δ *be a constraint set,* T *a type environment and E an expression then:*

$$\mathrm{sse}_\omega(\mathrm{T},E) \not\subseteq \mathrm{sse}_\Delta(\mathrm{T},E)$$

Proof We just have to provide a counter-example. Assume that **x** is defined.

```
E ≡ ((lambda (f g) g x)
     e
     (rec (f y) (f y)))

e ≡ (rec (f i) (f i))
```

It's easy to check that e is in $\mathrm{sse}_\omega(\mathrm{T},E)$ but not in $\mathrm{sse}_\Delta(\mathrm{T},E)$. □

7 Minimal Time

This section should be called *Minimal Time and Constraint Set* as we show that, when an expression is type correct, it has a minimal time and also a minimal constraint set (given the minimal time).

Informally, we define an ordering on type judgments using an instantiation relation. Hence, We can state a theorem (Th. 35) about a minimal time. It appears that, if this minimal type judgment is not unique, all the minimal type judgments are equivalent (through the relation $\simeq$). This set of minimal judgments can be ordered with an inclusion relation on constraint sets. With this newly defined order, we can state a theorem about a minimal constraint set (Th. 32). Fortunately, although the minimal type judgment (w.r.t. the constraint set) is not uniquely determined, all such judgments are α-equivalent (i.e. equivalent through α-renaming).

In practice, we first, define the order constraint set and, second, define the order on time. We start with the function Ord used to extract the set of order variables from various objects.

Definition 24 (Ord) *We write* $\mathrm{Ord}(X)$ *for the set of order variables present in the object* X*:*

$$\begin{aligned}
\mathrm{Ord}(\Delta) &= \{M \mid \exists\{m \sqsubset m'\} \in \Delta \wedge (m \sim M \vee m' \sim M)\} \\
\mathrm{Ord}(m) &= \{M \mid m \sim M \oplus m'\} \\
\mathrm{Ord}(t) &= \{M \mid t \sim t_1 \xrightarrow{m} t_0 \ \wedge (M \in \mathrm{Ord}(m) \vee M \in \mathrm{Ord}(t_1) \vee M \in \mathrm{Ord}(t_0))\} \\
\mathrm{Ord}(\mathrm{T}) &= \{M \mid \exists i \in \mathrm{dom}(\mathrm{T}) \wedge M \in \mathrm{Ord}(\mathrm{T}(i))\}
\end{aligned}$$

Valid judgments stay valid when a substitution on order variables is applied.

Lemma 25 *If the judgment* $\Delta, \mathrm{T} \vdash e : t \$ m$ *holds, then for all substitution* S *that map order variables from* $\mathrm{Ord}(\Delta, \mathrm{T} \vdash e : t \$ m)$ *to times, we know that* $S(\Delta), S(\mathrm{T}) \vdash e : S(t) \$ S(m)$ *holds.*

Proof By structural induction on the expression e. □

We also need to define the set of order variables and the set of significant order variables of a judgment. Intuitively, an order variable is significant if it is required by a static rule during typing.

Definition 26

$$\begin{aligned}
\mathrm{Ord}(\Delta, \mathrm{T} \vdash e : t \$ m) &= \mathrm{Ord}(\Delta) \cup \mathrm{Ord}(\mathrm{T}) \cup \mathrm{Ord}(t) \cup \mathrm{Ord}(m) \\
\mathrm{Sig}(\Delta, \mathrm{T} \vdash e : t \$ m) &= \mathrm{Ord}(\mathrm{T}|_{FV(e)}) \cup \mathrm{Ord}(t) \cup \mathrm{Ord}(m)
\end{aligned}$$

Some judgments are equivalent up to an α-renaming.

Definition 27 ($\equiv$) *A type judgment* $\Delta, \mathrm{T} \vdash e : t \$ m$ *is* α*-equivalent (equal up to an* α*-renaming) to another* $\Delta', \mathrm{T}' \vdash e : t' \$ m'$*, written* $\Delta, \mathrm{T} \vdash e : t \$ m \equiv \Delta', \mathrm{T}' \vdash e : t' \$ m'$*, if there exists a renaming (a one-to-one map),* γ*, from* $\mathrm{Ord}(\Delta, \mathrm{T} \vdash e : t \$ m)$ *to* $\mathrm{Ord}(\Delta', \mathrm{T}' \vdash e : t' \$ m')$*. Then, we have:*

$$\begin{aligned}
\gamma(\Delta) &= \Delta' \qquad & \gamma(t) &\sim t' \\
\gamma(\mathrm{T}) &= \mathrm{T}' \qquad & \gamma m &\sim m'
\end{aligned}$$

The inclusion relation on judgments is the extension of the inclusion on constraint sets. The inclusion on constraint sets is the standard set inclusion. In fact, to get a more general relation, we also allow the inclusion on type environments.

Definition 28 ($\sqsubseteq$) *A type judgment $\Delta, \mathrm{T} \vdash e : t \$ m$ is included in another $\Delta', \mathrm{T}' \vdash e : t' \$ m'$, written $\Delta, \mathrm{T} \vdash e : t \$ m \sqsubseteq \Delta', \mathrm{T}' \vdash e : t' \$ m'$, if there is a type judgment $\Delta'', \mathrm{T}'' \vdash e : t'' \$ m''$ such that:*

$$\begin{cases} \Delta'', \mathrm{T}'' \vdash e : t'' \$ m'' \equiv \Delta', \mathrm{T}' \vdash e : t' \$ m' & \\ \Delta \subseteq \Delta'' & t \sim t'' \\ \mathrm{T} \subseteq \mathrm{T}'' & m \sim m'' \end{cases}$$

When two judgment are included in each other, they are α-equivalent.

Lemma 29 *Let $\Delta, \mathrm{T} \vdash e : t \$ m$ and $\Delta', \mathrm{T}' \vdash e : t' \$ m'$ be two valid type judgments. Then, the following implication holds:*

$$\left.\begin{array}{l} \Delta, \mathrm{T} \vdash e : t \$ m \sqsubseteq \Delta', \mathrm{T}' \vdash e : t' \$ m' \\ \Delta', \mathrm{T}' \vdash e : t' \$ m' \sqsubseteq \Delta, \mathrm{T} \vdash e : t \$ m \end{array}\right\} \Longrightarrow \Delta, \mathrm{T} \vdash e : t \$ m \equiv \Delta', \mathrm{T}' \vdash e : t' \$ m'$$

Proof From the hypothesis, we know there are two valid type judgments $\Delta'', \mathrm{T}'' \vdash e : t'' \$ m''$ and $\Delta''', \mathrm{T}''' \vdash e : t''' \$ m'''$ such that:

$$\begin{cases} \Delta'', \mathrm{T}'' \vdash e : t'' \$ m'' \equiv & \Delta', \mathrm{T}' \vdash e : t' \$ m' \\ \Delta''', \mathrm{T}''' \vdash e : t''' \$ m''' \equiv & \Delta, \mathrm{T} \vdash e : t \$ m \\ \Delta \subseteq \Delta'' & \Delta' \subseteq \Delta''' \\ \mathrm{T} \subseteq \mathrm{T}'' & \mathrm{T}' \subseteq \mathrm{T}''' \\ t \sim t'' & t' \sim t''' \\ m \sim m'' & m' \sim m''' \end{cases}$$

Hence, there are two one-to-one maps γ and γ' such that:

$$\begin{cases} \gamma(\Delta'') = \Delta' & \gamma'(\Delta) = \Delta''' \\ \gamma(\mathrm{T}'') = \mathrm{T}' & \gamma'(\mathrm{T}) = \mathrm{T}''' \\ \gamma(t'') \sim t' & \gamma'(t) \sim t''' \\ \gamma(m'') \sim m' & \gamma'(m) \sim m''' \end{cases}$$

From $m \sim m''$ we know that $\gamma(m) \sim \gamma(m'') \sim m'$. In the same way, we know that $\gamma(t) \sim \gamma(t'') \sim t'$. From $\Delta \subseteq \Delta''$ and $\Delta' \subseteq \Delta'''$, we know that $\gamma(\Delta) \subseteq \gamma(\Delta'') = \Delta'$ and that $\Delta' \subseteq \gamma'(\Delta)$. Hence we have the embedding of Δ' in the two images of Δ, i.e. $\gamma(\Delta) \subseteq \Delta' \subseteq \gamma'(\Delta)$. As γ and γ' are one-to-one maps, the cardinalities of the three constraint sets are equal and $\gamma(\Delta) = \Delta'$. By a similar argument, we can prove that $\gamma(\mathrm{T}) = \mathrm{T}'$. The existence of map γ proves the equivalence between $\Delta, \mathrm{T} \vdash e : t \$ m$ and $\Delta', \mathrm{T}' \vdash e : t' \$ m'$. □

We now define the equivalence relation between judgments. The difference with the α-equivalence is that we require only the constraint sets (respectively the type environments) to be equal on significant order variables of the type judgment (respectively on free variables of the expression).

Definition 30 ($\simeq$) *A type judgment $\Delta, \mathrm{T} \vdash e : t \$ m$ is equivalent to another $\Delta', \mathrm{T}' \vdash e : t' \$ m'$, written $\Delta, \mathrm{T} \vdash e : t \$ m \simeq \Delta', \mathrm{T}' \vdash e : t' \$ m'$, if there exists a map γ, of order variables from* $\mathrm{Ord}(\Delta, \mathrm{T} \vdash e : t \$ m)$ *to* $\mathrm{Ord}(\Delta', \mathrm{T}' \vdash e : t' \$ m')$ *that is a renaming (one-to-one map) on* $\mathrm{Sig}(\Delta, \mathrm{T} \vdash e : t \$ m)$. *We have:*

$$\begin{array}{ll} \gamma(\Delta|_{Sig(\Delta, \mathrm{T} \vdash e : t \$ m)}) = \Delta' & \gamma(t) \sim t' \\ \gamma(\mathrm{T}|_{FV(e)}) = \mathrm{T}' & \gamma m \sim m' \end{array}$$

Included judgments are equivalents.

Lemma 31 *Let* $\Delta, T \vdash e : t \$ m$ *and* $\Delta', T' \vdash e : t' \$ m'$ *be two valid type judgments. Then, the following implication holds:*

$$\Delta, T \vdash e : t \$ m \sqsubseteq \Delta', T' \vdash e : t' \$ m' \Longrightarrow \Delta, T \vdash e : t \$ m \simeq \Delta', T' \vdash e : t' \$ m'$$

Proof From the hypothesis, we know there are a renaming γ of order variables from $Ord(\Delta', T' \vdash e : t' \$ m')$ to $Ord(\Delta'', T'' \vdash e : t'' \$ m'')$ and a valid type judgment $\Delta'', T'' \vdash e : t'' \$ m''$ such that:

$$\begin{cases} \Delta'', T'' \vdash e : t'' \$ m'' \equiv \Delta', T' \vdash e : t' \$ m' \\ \gamma(\Delta') = \Delta'' \supseteq \Delta \\ \gamma(T') = T'' \supseteq T \\ \gamma(t') \sim t'' \sim t \\ \gamma(m') \sim m'' \sim m \end{cases}$$

The map γ is a renaming on $Ord(\Delta, T \vdash e : t \$ m)$ and hence it is renaming on $Sig(\Delta, T \vdash e : t \$ m)$. The inclusion of T in T'' is only induced by bindings present in T'' and not T. As these bindings are not in $FV(e)$, we can conclude $\gamma(T'|_{FV(e)}) = T|_{FV(e)}$. By a similar argument about the constraints present in Δ', we can prove that $\gamma(\Delta'|_{Sig(\Delta', T' \vdash e : t' \$ m')}) = \Delta|_{Sig(\Delta, T \vdash e : t \$ m)}$. □

We can now state the minimal constraint set theorem. It says that in a set of equivalent judgments, there is a unique judgment (up to an α-renaming) that is included in all the other judgments.

Theorem 32 (Minimal Constraint Set) *Let* Δ *be a constraint set,* T *a type environment, e an expression, t a type and m a time. If* $\Delta, T \vdash e : t \$ m$ *is a valid judgment, then there exists a valid judgment* $\Delta_\Delta, T_\Delta \vdash e : t_\Delta \$ m_\Delta$ *such that:*

(1) $\Delta_\Delta, T_\Delta \vdash e : t_\Delta \$ m_\Delta \sqsubseteq \Delta, T \vdash e : t \$ m$
(2) All other type judgments equivalent to $\Delta, T \vdash e : t \$ m$ *include* $\Delta_\Delta, T_\Delta \vdash e : t_\Delta \$ m_\Delta$.

Proof See the section on minimal constraint sets in [FM89]. □

Now, we have to show that, among all the valid judgments for an expression, there exists a unique (up to the equivalence relation) judgment more general than all the others. First, we need a definition for the notion of instantiation.

Definition 33 (Instance) *A type judgment* $\Delta', T' \vdash e : t' \$ m'$ *is an instance of another* $\Delta, T \vdash e : t \$ m$, *written* $\Delta, T \vdash e : t \$ m \preceq \Delta', T' \vdash e : t' \$ m'$ *if there exists a substitution* γ *from orders to orders such that:*

$$\begin{cases} \gamma(\Delta) \subseteq \Delta' \\ \gamma(T|_{FV(e)}) = T'|_{FV(e)} \\ \gamma(t) \sim t' \\ \Delta' \vdash \gamma(m) \sqsubset m' \end{cases}$$

When two judgments are both instantiations of each other, they are equivalent (under the $\simeq$ relation).

Lemma 34 *Let* $\Delta, T \vdash e : t \$ m$ *and* $\Delta', T' \vdash e : t' \$ m'$ *be two valid type judgments. Then the following implication holds:*

$$\left.\begin{array}{l} \Delta, T \vdash e : t \$ m \preceq \Delta', T' \vdash e : t' \$ m' \\ \Delta', T' \vdash e : t' \$ m' \preceq \Delta, T \vdash e : t \$ m \end{array}\right\} \Longrightarrow \Delta, T \vdash e : t \$ m \simeq \Delta', T' \vdash e : t' \$ m'$$

Proof From the hypothesis, there are two substitutions γ and γ' such that:

$$\begin{cases} \gamma(\Delta) \subseteq \Delta' & \gamma'(\Delta') \subseteq \Delta \\ \gamma(\mathrm{T}|_{FV(e)}) = \mathrm{T}'|_{FV(e)} & \gamma'(\mathrm{T}'|_{FV(e)}) = \mathrm{T}|_{FV(e)} \\ \gamma(t) \sim t' & \gamma'(t') \sim t \\ \Delta' \vdash \gamma(m) \sqsubset m' & \Delta \vdash \gamma'(m') \sqsubset m \end{cases}$$

First, we know that for all i in $\mathrm{FV}(e)$, $\gamma(\mathrm{T}(i)) \sim \mathrm{T}'(i)$, $\gamma'(\mathrm{T}'(i)) \sim \mathrm{T}(i)$ and $\gamma'(\gamma(\mathrm{T}(i))) \sim \mathrm{T}(i)$. Now, from $\Delta' \vdash \gamma(m) \sqsubset m'$, we know that $\gamma'(\Delta') \vdash \gamma'(\gamma(m)) \sqsubset \gamma'(m')$ and then $\gamma'(m')$ is embedded between $\gamma'(\gamma(m))$ and m, i.e. $\Delta \vdash \gamma'(\gamma(m)) \sqsubset \gamma'(m') \sqsubset m$. But m is a time expression built from latent times of free variables of e. These latent times are invariant under $\gamma' \circ \gamma$, so $\gamma'(\gamma(m)) \sim m$ and $\gamma'(m') \sim m$. Note that $\gamma' \circ \gamma$ is a one-to-one map on $\mathrm{Sig}(\Delta, \mathrm{T} \vdash e : t \$ m)$, and hence γ and γ' are also one-to-one maps on $\mathrm{Sig}(\Delta, \mathrm{T} \vdash e : t \$ m)$ and $\mathrm{Sig}(\Delta', \mathrm{T}' \vdash e : t' \$ m')$ respectively. The equality of $\gamma(\Delta)$ and Δ' on $\mathrm{Sig}(\Delta, \mathrm{T} \vdash e : t \$ m)$ is induced by the inclusion. □

Finally, we can state the minimal time theorem.

Theorem 35 (Minimal Time) *Let Δ be a constraint set, T a type environment, e an expression, t a type and m a time. If $\Delta, \mathrm{T} \vdash e : t \$ m$ is a valid judgment then there is a valid judgment $\Delta_*, \mathrm{T}_* \vdash e : t_* \$ m_*$ such that:*

(1) $\Delta_*, \mathrm{T}_* \vdash e : t_* \$ m_* \preceq \Delta, \mathrm{T} \vdash e : t \$ m$
(2) All other valid judgments are instances of $\Delta_*, \mathrm{T}_* \vdash e : t_* \$ m_*$

Proof See [FM88, FM90]. □

8 Conclusion

We have presented a time system, i.e. a type system especially designed to provide expected run-time estimates. We have shown that it is more powerful than the previous proposed time systems. It also enjoys a minimal time property. This time system is the first attemp to merge as a whole the framework of the *effect system* from Lucassen and Gifford [LG88] with the subtyping techniques of Fuh and Mishra [FM88]. This allows us to detect and isolate each basic recursion in a program written using a realistic language (with higher-order functions and side-effects). Note that the time system presented here provides maximal time estimates but the same principle may be used to give minimal times.

Acknowledgments

Many thanks to many people. To Pierre Jouvelot for his comments and his help with the related work section. To Martin Odersky who gave me the spark, the starting point of this time system. To Mark Jones who was a great help for the minimal type property. To Kung Chen for his comments on technical points in the theorems and in the proofs. And finally, to Paul Hudak, Martin Odersky and the Haskell research group who have welcomed me at Yale during the past three months.

Bibliography

[AH87] Abramsky S. and Hankin C., *Abstract Interpretation of Declarative Languages. Ellis Horwood Limited, Chichester, England* (1987).

[B78] Backus J. W., Can Programming be Liberated from Von Neumann Style? A Functionnal Style and its Algebra of Programs. *CACM 21,* 8 (August 1978), 613-641.

[C90] Consel C., Binding Time Analysis for Higher-Order Untyped Functional Languages. *LFP'90, Nice, ACM,* (June 1990), 264-273.

[DP90] Davey B. A. and Priestley H. A., *Introduction to Lattices and Order. Cambridge Math. Textbooks, Cambridge University Press* (1990).

[D92] Dornic V., *Analyse de Complexité des Programmes : Vérification et Inférence. PhD Thesis University Paris VI,* Ecole des Mines, Tech Report, EMP-CRI A/212 (June 1992).

[DJG91] Dornic V., Jouvelot P. and Gifford D. K., Polymorphic Time Systems for Estimating Program Complexity. *WSA'91, Bordeaux, France,* (October 1991). *ACM LoPLaS, Letter on Programming Languages and Systems 1,* 1 (March 1992).

[FV87] Flajolet P. and Vitter J. S., Average-Case Analysis of Algorithms and Data Structures. *Research report, INRIA 718,* (August 1987).

[FM88] Fuh Y-C. and Mishra P., Type Inference with Subtypes. *ESOP'88, Lecture Notes in Computer Science 300,* (1988) 94-114.

[FM89] Fuh Y-C. and Mishra P., Polymorphic Subtype Inference: Closing the Theory-Practice Gap. *TaPSoft'89, Barcelona, Spain, LNCS 352,* (March 1989) 167-183.

[FM90] Fuh Y-C. and Mishra P., Type Inference with Subtypes. *Theoretical Computer Science 73, North-Holland,* (1990) 155-175.

[G86] Gifford D. K., Integrating Functional and Imperative Programming. *ACM Conference on Lisp and Functional Programming,* (August 1986) 28-38.

[G83] Gray S. L., Using Futures to Exploit Parallelism in Lisp. *MIT SB Master Thesis,* (1983).

[GSTRG92] Grundman D., Stata R., O'Toole J. W., Reistad B. and Gifford D. K., μFX-a/DLX - A Pedagogic Compiler. *MIT-LCS Technical Report,* (1992).

[HMT89] Harper R., Milner R. and Tofte M., The definition of Standard ML. Version 3. *LFCS Report, DCS, University of Edinburgh,* (May 1989).

[HC88] Hickey T. and Cohen J., Automating Program Analysis. *Journal of the ACM 35,* 1 (January 1988) 185-220.

[JG91] Jouvelot P. and Gifford D. K., Algebraic Reconstruction of Types and Effects. *Proceedings of the ACM PoPL'91 Conference, Orlando Fl.,* (January 1991).

[K81] Kozen D., Semantics of Probabilistic Programs. *J. Comput. Syst. Sci. 22* (1981), 328-350.

[L87] Le Métayer D., Analysis of Functional Programs by Program Transformation. *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium, North-Holland, Amsterdam,* (1987).

[L88] Le Métayer D., ACE: An Automatic Complexity Evaluator. *ACM TOPLAS, Transaction on Programming Languages and Systems 10,* 2 (April 1988), 248-266.

[LG88] Lucassen J. M. and Gifford D. K., Polymorphic Effect Systems. *PoPL'88, San Diego, ACM,* (January 1988).

[M79] McCracken N. J., An Investigation of a Programming Language with a Polymorphic Type Structure. *PhD Dissertation, Syracuse University,* (1979).

[R79] Ramshaw L. H., Formalizing the Analysis of Algorithms. *Report SL-79-5, Xerox Palo Alto Research Center, Palo Alto, Calif,* (1979).

[R65] Robinson J. A., A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM 12,* 1 (1965) 23-41.

[R86] Rosendahl M., Automatic Program Analysis. *Master's Thesis. Institute of Datalogy, University of Copenhagen,* (1986).

[R89] Rosendahl M., Automatic Complexity Analysis. *Proceedings of FPCA'89, ACM,* (1989) 144-156.

[S88] Sands D., Complexity Analysis for Higher Order Languages. *Report DOC 88/14, Imperial College, London,* (October 1988).

[S90] Sands D., Complexity Analysis for a Lazy High-Order Languages. *Lecture Notes in Computer Science, ESOP'90,* (1990).

[TJ91] Talpin J-P. and Jouvelot P., On Reconstructing Type, Effect and Region in Polymorphic Functional Languages and its Applications. *Research Report E-149, Ecole des Mines de Paris, Presented at the CPC'91 workshop under the title* Applications of types and effect inference, (February 1991).

[TJ91] Talpin J-P. and Jouvelot P., Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming 2,* 3 (1992) 245-272.

[T36] 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).

[W88] Wadler P., Strictness Analysis Aids Time Analysis. *PoPL'88, San Diego, ACM* (January 1988)

[W75] Wegbreit B., Mechanical Program Analysis. *Communication of the ACM 18,* 9 (September 1975), 528-539