Extracting Feasible Programs

Jean-Yves Marion
Loria, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France.
Jean-Yves.Marion@loria.fr,
http://www.loria.fr/∼marionjy

Abstract

This work in progress paper presents a methodology for reasoning about the computational complexity of functional programs, which are extracted from proofs.

We suggest a first order arithmetic $\mathbf{AT}^0$ which is a syntactic restriction of Peano arithmetic. We establish that the set of functions which is provably total in $\mathbf{AT}^0$, is exactly the set of polynomial time functions. This result has been accepted at Conference of the European Association for Computer Science Logic (CSL), 2001.

Compared to others feasible arithmetics, $\mathbf{AT}^0$ is surprisingly simpler. The main feature of $\mathbf{AT}^0$ concerns the treatment of the quantification. The range of quantifiers is restricted to the set of *actual terms* which is the set of constructor terms with variables. The second feature concerns a restriction of the inductive formulas.

Although this paper addresses some theoretical aspects of program extractions, it is relevant for practical issue, in the long term, because certifying the computational resource consumed by a program is a challenging issue.

Introduction

From Heyting's semantic and the Curry-Howard isomorphism, we know how to extract a program from a proof whose denotation is the conclusion of the proof. Several systems rely on this concept, like Nuprl and Coq. Those systems allow one to construct a correct program from a specification. However, the efficiency of the program obtained is not guaranteed. But, efficiency is a crucial property of a running implementation. Constable pointed out this practical issue. Benzinger (Benzinger 1999) has developed a prototype *ACA* to determine the runtime of Nuprl-extracted programs. For this, *ACA* determines an upper bound on the number of reduction steps of a Nuprl program by solving recurrence equations. There are several other related approaches, see for example (Métayer 1988; Flajolet, Salvy, & Zimmermann 1990; Crary & Weirich 2000), which are based on counting computational resources.

Here, we rather propose a proof theoretical method to analyse the runtime complexity of extracted programs. We shall briefly describe the system $\mathbf{AT}^0$ in the next Section for which the set of provably total functions is exactly the set of polynomial time functions. For a full description of $\mathbf{AT}^0$, consult (Marion 2001).

Up to now, $\mathbf{AT}^0$ can prove the termination of first order functional programs which are simply based on primitive recursion templates. However, $\mathbf{AT}^0$ is an open system in which we can deal with other data-structures and we expect to shortly incorporate other induction schema, which encompass other algorithm patterns, and deal with higher type constructions.

The goal of the long term research is to provide a framework to provide an analysis of an extracted program, from which we can (i) deduce the complexity of the program and (ii) also have hints to automatically transform the extracted program into a more efficient one. Let us explain what we mean by considering first order functional programming ar rewrite rules. We have used termination proofs in (Marion 2000; Marion & Moyen 2000) to answer partially to (i) and (ii). For this, we have defined resctrictions of a termination ordering, in such a way that a terminating program p computes a polynomial time function. The interesting point is that the p can be evaluated in exponential time (in the number of reduction steps), but denotes a polynomial time function. This happens with recursive specifications of problems which are efficiently solved by dynamic programming. In our case, the termination proof gives information to construct a call by value interpreter for p with a cache. The cache memorizes the value of recursive call which shall be used later on. Moreover, because of the termination proof, we know how to maintain the smallest possible cache.

In other words, the goal (or the dream) of this research is to extract a correct and efficient program from the termination proofs with guaranted complexity.

Let us finish this discusion by wondering why the extraction of efficient programs is a difficult task. The traditional meta-theory of program reasoning fails to deal with such a question because analysing a specification to extract a "good" algorithm is an intentional property. On the other hand, Theory of Computational Complexity theory (TCC) delineates classes of functions, which are computable with, bounded resources. TCC characterisations are extensional, that is all functions of a given class are captured, but most of the "good" algorithms are missing. Runtime analysis of programs necessitates to reason on programs, or on proofs

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

in the "proofs-as-programs" context. For this, we need to develop logics to study algorithmic contents of proofs.

A feasible arithmetic

Terms are divided into different categories which are listed in Figure 1. *Actual terms* are built up from constructors of $\mathcal{W}$, and variables of $\mathcal{X}$, and forms the set $\mathcal{T}(\mathcal{W},\mathcal{X})$. The logical rules of $\mathbf{AT}$ are written in Figure 2.

The difference with the $\{\rightarrow,\wedge,\forall\}$-fragment of the minimal logic is *the actual elimination quantifier principle* which is obtained by the $\forall E^{\mathbf{S}}$-rule.

Now, we extend $\mathbf{AT}$ to an arithmetic $\mathbf{AT}(\mathbf{W})$ in order to reason about the free word algebra $\mathcal{T}(\mathcal{W})$. The set of words is denoted by a unary predicate $\mathbf{W}$ together with the rules displayed in Figure 3.

Following Martin-Löf (Martin-Löf 1984) and Leivant (Leivant 1994a; 2000), the introduction rules indicate the construction of words, and the elimination rules specify the computational behaviour associated with them. Both elimination rules, that is the induction rule and the selection rule, are necessary, because of the actual elimination quantifier principle. Indeed, the induction rule schema Ind($\mathbf{W}$) corresponds to the usual induction. However, the range of the universal quantifier is restricted to actual terms. So, the last quantifier of the induction filters the instantiation through the $\forall E^{\mathbf{S}}$-rule. Roughly speaking, an induction is guarded by a universal quantifier, like a proof-net box in linear logic.

On the other hand, the selection rule expresses that a word t is either the empty word $\boldsymbol{\epsilon}$, or $\mathbf{s_i}(y)$ for some term y. We shall employ the selection rule to perform definitions by cases over words. Unlike the induction rule, the term t in the conclusion of the selection rule can be any term. It is worth noticing that the application of the selection rule is restricted. There is no application of $\forall E^{\mathbf{S}}$-rule in the derivations $\pi_{\mathbf{s_0}}$ and $\pi_{\mathbf{s_1}}$. Thus, we prohibit nested applications of induction rule, inside the selection rule. Otherwise it would be possible to unguard an induction.

Reasoning over programs

An equational program $\mathtt{f}$ is a set of (oriented) equations $\mathcal{E}$. Each equation is of the form $\mathtt{f}(p_1,\cdots,p_n)\rightarrow t$ where each p_i is an actual term, and corresponds to a pattern. The term t is in $\mathcal{T}(\mathcal{W},\mathcal{F},\mathcal{X})$ and each variable of t also appears in $\mathtt{f}(p_1,\cdots,p_n)$.

Definition A confluent equational program $\mathtt{f}$ computes a function $[\![\mathtt{f}]\!]$ over $\mathcal{T}(\mathcal{W})$ which is defined as follows.
For each $\mathbf{w}_i,\mathbf{v}\in\mathcal{T}(\mathcal{W})$, $[\![\mathtt{f}]\!](\mathbf{w}_1,\cdots,\mathbf{w}_n)=\mathbf{v}$ iff the normal form of $\mathtt{f}(\mathbf{w}_1,\cdots,\mathbf{w}_n)$ is $\mathbf{v}$, otherwise $[\![\mathtt{f}]\!](\mathbf{w}_1,\cdots,\mathbf{w}_n)$ is undefined.

Let $\mathtt{f}$ be an equational program. We define $\mathbf{AT}(\mathtt{f})$ as the calculus $\mathbf{AT}(\mathbf{W})$ extended with the replacement rule below,

$$\frac{A[u\theta]}{A[v\theta]}\,\mathrm{R}$$

where $(v\rightarrow u)\in\mathcal{E}$ and θ is a substitution $\mathcal{X}\rightarrow\mathcal{T}(\mathcal{W},\mathcal{X})$.

Definition A function ϕ of arity n is provably total in $\mathbf{AT}(\mathbf{W})$ iff there are an equational program $\mathtt{f}$ such that $\phi=[\![\mathtt{f}]\!]$ and a derivation in $\mathbf{AT}(\mathtt{f})$ of

$$\mathrm{Tot}(\mathtt{f})\equiv\forall x_1\cdots x_n.\mathbf{W}(x_1),\cdots,\mathbf{W}(x_n)\rightarrow\mathbf{W}(\mathtt{f}(x_1,\cdots,x_n))$$

Definition A formula $A[x]$ is an induction formula if $\forall x.\mathbf{W}(x)\rightarrow A[x]$ is the conclusion of an induction. Define $\mathbf{AT}^0(\mathbf{W})$ as the restriction of $\mathbf{AT}(\mathbf{W})$ in which induction formulas are just conjunctions of predicates (i.e. atomic formulas).

Theorem [Main result] A function ϕ is polynomial time computable if and only if the function ϕ is provably total in $\mathbf{AT}^0$.

Example We begin with the word concatenation whose equations are

$$\begin{aligned}\mathtt{cat}(\boldsymbol{\epsilon},w)&\rightarrow w\\ \mathtt{cat}(\mathbf{s_i}(x),w)&\rightarrow\mathbf{s_i}(\mathtt{cat}(x,w))\qquad \mathbf{i=0,1}\end{aligned}$$

The derivation $\pi_{\mathtt{cat}}$ in Fiugre 4 shows that the concatenation is a provably total function of $\mathbf{AT}(\mathtt{cat})$.

Notice that the term w is any term, and so w can be substituted by a non-actual term. Let us investigate the word multiplication whose equations are

$$\begin{aligned}\mathtt{mul}(\boldsymbol{\epsilon},x)&\rightarrow\boldsymbol{\epsilon}\\ \mathtt{mul}(\mathbf{s_i}(y),x)&\rightarrow\mathtt{cat}(x,\mathtt{mul}(y,x))\qquad \mathbf{i=0,1}\end{aligned}$$

The word multiplication is a provably total function as the derivation in Figure 5 shows it.

Now, consider the equations defining the exponential :

$$\begin{aligned}\mathtt{exp}(\boldsymbol{\epsilon})&\rightarrow\mathbf{s_0}(\boldsymbol{\epsilon})\\ \mathtt{exp}(\mathbf{s_i}(y))&\rightarrow\mathtt{cat}(\mathtt{exp}(y),\mathtt{exp}(y))\qquad \mathbf{i=0,1}\end{aligned}$$

In order to establish that the program $\mathtt{exp}$ defines a provably total function, we have to make an induction. At the induction step, under the assumptions $\mathbf{W}(\mathtt{exp}(y))$ and $\mathbf{W}(y)$, we have to prove $\mathbf{W}(\mathtt{cat}(\mathtt{exp}(y),\mathtt{exp}(y)))$. However, $\mathtt{exp}(y)$ is not an actual term, and so we can not "plug in" the derivation $\pi_{\mathtt{cat}}$ to conclude.

Related works

One of the main advantage of the system $\mathbf{AT}^0$ compare to other approaches is that $\mathbf{AT}^0$ is conceptually simpler. Theories of feasible mathematics originate with Buss (Buss 1986) on bounded arithmetic. Subsequently, Leivant (Leivant 1991) established that the functions provably total in second order arithmetic with the comprehension axiom restricted to positive existential formulas, are exactly the polynomial time functions. Leivant (Leivant 1994a) also translated his characterisation (Leivant 1994b) of feasible functions by mean of ramified recursion. For this, he has introduced a sequence of predicate $\mathbf{N}_0,\mathbf{N}_1,\ldots$ corresponding to copies of $\mathbf{N}$ with increasing computational potential. Çağman, Ostrin and Wainer (Çağman, Ostrin, & Wainer 2000) defined

(Constructors)	$\mathcal{W} \ni \mathbf{c}$	$::=$	$\boldsymbol{\epsilon} \mid \mathbf{s_0} \mid \mathbf{s_1}$
(Function symbols)	$\mathcal{F} \ni \mathsf{f}$	$::=$	$\mathsf{f} \mid \mathsf{g} \mid \mathsf{h} \mid \ldots$ with fixed arities
(Variables)	$\mathcal{X} \ni x$	$::=$	$x \mid y \mid z \mid \ldots$
(Words)	$\mathcal{T}(\mathcal{W}) \ni \mathbf{w}$	$::=$	$\boldsymbol{\epsilon} \mid \mathbf{s_0}(\mathbf{w}) \mid \mathbf{s_1}(\mathbf{w})$
(Terms)	$\mathcal{T}(\mathcal{W}, \mathcal{F}, \mathcal{X}) \ni t$	$::=$	$\boldsymbol{\epsilon} \mid \mathbf{s_0}(t) \mid \mathbf{s_1}(t) \mid \mathsf{f}(t_1, \ldots, t_n) \mid x$
(Actual terms)	$\mathcal{T}(\mathcal{W}, \mathcal{X}) \ni p$	$::=$	$\boldsymbol{\epsilon} \mid \mathbf{s_0}(p) \mid \mathbf{s_1}(p) \mid x$

Figure 1: Categories of terms

Premiss : A (Predicate A)

Introduction rules	*Elimination rules*
$\dfrac{\begin{array}{c}\{A\}\\ \vdots \\ B\end{array}}{A \to B} \to I$	$\dfrac{A \to B \qquad A}{B} \to E$
$\dfrac{A_1 \qquad A_2}{A_1 \wedge A_2} \wedge I$	$\dfrac{A_1 \wedge A_2}{A_j} \wedge E$
$\dfrac{A}{\forall x.A} \forall I$	$\dfrac{\forall x.A}{A[x \leftarrow p]} \forall E^{\mathbf{S}}$, *where* $p \in \mathcal{T}(\mathcal{W}, \mathcal{X})$

Restrictions on the rules

- In $\forall I$-rule, x is not free in any premiss.
- In $\forall E^{\mathbf{S}}$-rule, p is an actual term, *i.e.* $p \in \mathcal{T}(\mathcal{W}, \mathcal{X})$.

Figure 2: Logical rules of $\mathbf{AT}$.

Introduction rules

$$\frac{}{\mathbf{W}(\boldsymbol{\epsilon})}\,\boldsymbol{\epsilon} I \qquad \frac{\mathbf{W}(t)}{\mathbf{W}(\mathbf{s_0}(t))}\,\mathbf{s_0} I \qquad \frac{\mathbf{W}(t)}{\mathbf{W}(\mathbf{s_1}(t))}\,\mathbf{s_1} I$$

Elimination rules

Selection
$$\frac{\begin{array}{c}\vdots\,\pi_{\mathbf{s_0}}\\ \mathbf{W}(y) \to A[\mathbf{s_0}(y)]\end{array} \quad \begin{array}{c}\vdots\,\pi_{\mathbf{s_1}}\\ \mathbf{W}(y) \to A[\mathbf{s_1}(y)]\end{array} \quad A[\boldsymbol{\epsilon}] \quad \mathbf{W}(t)}{A[t]}\,\mathrm{Sel}(\mathbf{W})$$

Induction
$$\frac{\forall y.A[y], \mathbf{W}(y) \to A[\mathbf{s_0}(y)] \quad \forall y.A[y], \mathbf{W}(y) \to A[\mathbf{s_1}(y)] \quad A[\boldsymbol{\epsilon}]}{\forall x.\mathbf{W}(x) \to A[x]}\,\mathrm{Ind}(\mathbf{W})$$

Restrictions on the rules :

- In $\mathrm{Sel}(\mathbf{W})$-rule, derivations of $\pi_{\mathbf{s_0}}$ and $\pi_{\mathbf{s_1}}$ do not use the rule $\forall E^{\mathbf{S}}$. The variable y must not occur in any assumption on which $A[t]$ depends.

Figure 3: Rules for word reasonning in $\mathbf{AT}(\mathbf{W})$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\{\mathbf{W}(\mathtt{cat}(z,w))\}}{\mathbf{W}(\mathbf{s_i}(\mathtt{cat}(z,w)))}\,\mathbf{s_i}I
}{\mathbf{W}(\mathtt{cat}(\mathbf{s_i}(z),w))}\,\mathrm{R}
\quad \{\mathbf{W}(z)\}
}{\mathbf{W}(z),\mathbf{W}(\mathtt{cat}(z,w))\to\mathbf{W}(\mathtt{cat}(\mathbf{s_i}(z),w))}\to I
}{\forall z.\mathbf{W}(z),\mathbf{W}(\mathtt{cat}(z,w))\to\mathbf{W}(\mathtt{cat}(\mathbf{s_i}(z),w))}\,\forall I
\quad
\dfrac{\mathbf{W}(w)}{\mathbf{W}(\mathtt{cat}(\boldsymbol{\epsilon},w))}\,\mathrm{R}
}{\forall x.\mathbf{W}(x)\to\mathbf{W}(\mathtt{cat}(x,w))}\,\mathrm{Ind}(\mathbf{W})
$$

Figure 4: Concatenation

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\begin{array}{c}
\{\mathbf{W}(\mathtt{mul}(z,x))\}\\
\vdots\ \pi_{\mathtt{cat}}[w\leftarrow\mathtt{mul}(z,x)]\\
\forall x.\mathbf{W}(x)\to\mathbf{W}(\mathtt{cat}(x,\mathtt{mul}(z,x)))
\end{array}
}{\mathbf{W}(x)\to\mathbf{W}(\mathtt{cat}(x,\mathtt{mul}(z,x)))}\,\forall E^{\mathbf{S}}
\quad \{\mathbf{W}(x)\}
}{\mathbf{W}(\mathtt{cat}(x,\mathtt{mul}(z,x)))}\to E
}{\mathbf{W}(\mathtt{mul}(\mathbf{s_i}(z),x))}\,\mathrm{R}
}{\mathbf{W}(z),\mathbf{W}(\mathtt{mul}(z,x))\to\mathbf{W}(\mathtt{mul}(\mathbf{s_i}(z),x))}\to I
}{\forall z.\mathbf{W}(z),\mathbf{W}(\mathtt{mul}(z,x))\to\mathbf{W}(\mathtt{mul}(\mathbf{s_i}(z),x))}\,\forall I
\quad
\dfrac{\dfrac{}{\mathbf{W}(\boldsymbol{\epsilon})}\,\boldsymbol{\epsilon}I}{\mathbf{W}(\mathtt{mul}(\boldsymbol{\epsilon},x))}\,\mathrm{R}
}{
\dfrac{\forall y.\mathbf{W}(y)\to\mathbf{W}(\mathtt{mul}(y,x))}{\forall x.\forall y.\mathbf{W}(x),\mathbf{W}(y)\to\mathbf{W}(\mathtt{mul}(y,x))}\,(\forall E^{\mathbf{S}};\to I;\forall I;\forall I)
}
$$

Figure 5: Multiplication

a two sorted Peano arithmetic PA$(;)$ in the spirit of Bellantoni and Cook (Bellantoni & Cook 1992). They characterize the functions computable in linear space, and the elementary functions. Predicates have two kinds of arguments : safe and normal. Quantifiers are allowed only over safe terms and range over hereditary basic terms. In a recent article (Leivant 2001), Leivant suggests a new direction by giving some structural conditions on proof hipothesis and on inductive formulas. There are also theories of feasible mathematics which are affiliated to linear logic. Girard, Scedrov and Scott in (J.-Y. Girard 1992) have introduced bounded linear logic, in which resources are explicitly counted. Then, Girard (Girard 1998) constructed light linear logic which is a second order logic with a new modality which controls safely the resources. See also the works of Asperti (Asperti 1998) and Roversi (Roversi 1999). Lastly, Bellantoni and Hofmann (Bellantoni & Hofmann 2000) and Schwichtenberg (Schwichtenberg), have proposed feasible arithmetics based on linear logic with extra counting modalities.

Comments

These examples illustrate that actual terms play a role similar to terms of higher tier (safe) used in ramified recursions, as defined by Bellantoni and Cook (Bellantoni & Cook 1992), and Leivant in (Leivant 1994b). Intuitively, we do not assume that two terms are equal just because they have the same value. We are not concerned by term denotations, but rather by the resource necessary to evaluate a term, or in other words, by term intention. From this point of view, a non-actual term is unsafe. So, we have no justification to quantify over non-actual terms. On the other hand, there are no computation rules associated to actual terms, so they are safe with respect to polynomial-time computation. In a way, this idea is similar to “read-only” programs of Jones (Jones 1999).

The concept arising from the work of Simmons (Simmons 1988), Bellantoni and Cook (Bellantoni & Cook 1992) and Leivant (Leivant 1994b), is the ramification of the domain of computation and the ramification of recursion schemata. One usually compares this solution with Russell’s type theory. One unattractive feature is that objects are duplicated at different tiers. This drawback is eliminated here. It is amazing to see that this solution seems related to Zermelo or Quine answers to Russell’s type theory.

Lastly, the actual elimination quantifier principle reminds one of logic with existence predicate, in which quantifiers are supposed to range only over existing terms. The motivation is to take into account undefined terms. Such logics have their roots in works of Weyl (Weyl 1921) and Heyting (Heyting 1930), and have since extensively studied and are related to *free logic*.

References

Asperti, A. 1998. Light affine logic. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science(LICS’98)*, 300–308.

Bellantoni, S., and Cook, S. 1992. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity* 2:97–110.

Bellantoni, S., and Hofmann, M. 2000. A new ”feasible” arithmetic. *Journal of symbolic logic*. to appear.

Benzinger, R. 1999. *Automated complexity analysis of NUPRL extracts*. Ph.D. Dissertation, Cornell University.

Buss, S. 1986. *Bounded arithmetic*. Bibliopolis.

Çağman, N.; Ostrin, G.; and Wainer, S. 2000. Proof theoretic complexity of low subrecursive classes. In Bauer, F. L., and Steinbrueggen, R., eds., *Foundations of Secure Computation*, Foundations of Secure Computation, 249–286. IOS Press Amsterdam.

Crary, K., and Weirich, S. 2000. Ressource bound certification. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, 184 – 198.

Flajolet, P.; Salvy, B.; and Zimmermann, P. 1990. Automatic average-case analysis of algorithms. Technical Report 1233, INRIA.

Girard, J.-Y. 1998. Light linear logic. *Information and Computation* 143(2):175–204. prsent LCC’94, LNCS 960.

Heyting, A. 1930. Die formalen regeln der intuitionischen logik. *Sitzungsberichte der preussischen akademie von wissenschaften* 57–71.

J.-Y. Girard, A. Scedrov, P. S. 1992. A modular approach to polynomial-time computability. *Theoretical Computer Science* 97(1):1–66.

Jones, N. 1999. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science* 228:151–174.

Leivant, D. 1991. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS’91)*.

Leivant, D. 1994a. Intrinsic theories and computational complexity. In *Logical and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, 177–194. Springer.

Leivant, D. 1994b. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Clote, P., and Remmel, J., eds., *Feasible Mathematics II*. Birkhäuser. 320–343.

Leivant, D. 2000. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*. to appear.

Leivant, D. 2001. Termination proofs and complexity certification. In *Proceedings of the third international workshop on Implicit Computational Complexity*.

Marion, J.-Y., and Moyen, J.-Y. 2000. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, 25–42. Springer.

Marion, J.-Y. 2000. Analysing the implicit complexity of programs. *Information and Computation*. to appear.

Marion, J.-Y. 2001. Actual arithmetic and feasibility. In Fribourg, L., ed., *Computer Science Logic, Paris*, volume 2142 of *LNCS*, 115–129. Springer.

Martin-Löf, P. 1984. *Intuitionistic Type Theory*. Bibliopolis.

Métayer, D. L. 1988. Ace : an automatic complexity evaluator. *Transactions on programming languages and systems* 10(2).

Roversi, L. 1999. A P-Time completeness proof for light logics. In *Computer Science Logic, 13th International Workshop, CSL '99*, volume 1683 of *Lecture Notes in Computer Science*, 469–483.

Schwichtenberg, H. Feasible programs from proofs. http://www.mathematik.uni-muenchen.de/∼schwicht/.

Simmons, H. 1988. The realm of primitive recursion. *Archive for Mathematical Logic* 27:177–188.

Weyl, H. 1921. über die neue grundlagenkrise der mathematik. Mathematische zeitschrift.