

## Survey of Typed Assembly Language

The goal is to survey the development of Typed Assembly Language from system F and its practical application.

What's Typed Assembly Language (TAL)?

Traditional untyped assembly languages are extended with typing annotations, memory management primitives, and a sound set of typing rules.

Motivation of this survey (i.e., any advantage of TAL?)

- The memory safety, control flow safety, and type safety of TAL programs are guaranteed by the typing rules
- The typing constructs are so expressive that they can encode most source language programming features including arrays, higher-order, records, structures polymorphic functions and etc.
- Many low-level compiler optimizations are also permitted under the flexible TAL. Consequently, type-directed compilers will choose TAL as an ideal platform if they want to produce verifiably safe code for use in extensible operating system kernels or secure mobile code applications.

### Introduction

TAL has advantages mainly in two perspectives. Firstly, it helps reaping the benefits of types throughout the compiler.

Conventional untyped compiler < typed intermediate language < typed target language

Here the “<” sign indicates that the relationship among these three compilers (it's not so accurate using this sign, but it's more directly perceivable): comparing to conventional untyped compiler, a compiler with typed intermediate language has compelling advantages in compiling a source language. It provides necessary type information to many of the optimizations, such as CPS, closure conversion, unboxing, etc. And its type-checking ability also helps us with debugging. However, there is a conceptual line in all these compilers where type are lost, for ML, about 20% of the type information will get lost. We can solve this problem by extending typed intermediate languages into typed target languages, which is strongly TAL based on a generic RISC instruction set. And we don't need to worry about the corresponding type system, because the type system is surprisingly standard. It will support high-level ML like languages to generate well-typed and efficient code without affecting low-level optimization.

The second greatness of TAL is that it provides a fully automatic way to produce *proof carrying code*, so untrusted code and even potentially malicious code can be executed both efficiently and safely in a practical system. The idea of security is derived from Necula and Lee [2], however our approach provides a fully automatic procedure to generate TAL from a well-formed source term, which can't be achieved by Necula and Lee's .

## Overview

From polymorphic  $\lambda$ -calculus to TAL, the development of TAL is structured as four translations between five typed calculi

$$\lambda^F \xrightarrow{\text{CPS conversion}} \lambda^K \xrightarrow{\text{Closure conversion}} \lambda^C \xrightarrow{\text{Allocation}} \lambda^A \xrightarrow{\text{Code generation}} \text{TAL}$$

We will follow the development, go through the CPS conversion, closure conversion, explicit allocation and code generation translations, presenting the theoretical base and implementation of each step, showing that each of these translations take a well-typed source term and produce a well-typed target term. The whole picture is inspired from SML/NJ with exception that types are applied throughout the compilation [1]. Also the compiler used for System F to generate TAL through all these four intermediate  $\lambda$ -calculi is defined as judgment

$$\frac{\begin{array}{l} \vdash_F e: \tau \rightarrow P: \\ \vdash_F e: \tau \xrightarrow{\text{cps}} e_{\text{cps}} \quad \emptyset; \emptyset \vdash_K e_{\text{cps}} \xrightarrow{\text{clos}} e_{\text{clos}} \quad \vdash_C e_{\text{clos}} \xrightarrow{\text{hst}} P_{\text{hst}} \quad \vdash_H P_{\text{hst}} \xrightarrow{\text{alloc}} P_{\text{alloc}} \quad \vdash_A P_{\text{alloc}} \xrightarrow{\text{TAL}} P \end{array}}{\vdash_F e: \tau \rightsquigarrow P}$$

The illustration of development of will serve as the main part of this survey.

Step 1:

$$\lambda^F \xrightarrow{\text{CPS conversion}} \lambda^K$$

In this step, the objective is to complete CPS conversion, which means instead of having a function return a value, let a function take as an argument another function to which the result is given as an argument. We start from the source language  $\lambda^F$ , which is a call-by-value variant of the polymorphic  $\lambda$ -calculus augmented with product  $\langle \tau_1, \dots, \tau_n \rangle$  and recursively defined function  $\text{fix } x(x_1: \tau_1): \tau_2.e$ . And the operational semantics used to interpret  $\lambda^F$  are conventional call-by-value ones, the static semantic is a set of inference rules which will help us reach the judgments of the form  $\Delta; \Gamma \vdash_F e: \tau$ .

In order to have continuation passing style conversion achieved, all the intermediate computations are named and the need for a control stack is eliminated. And via function calls, all unconditional control transfers, function invocation and return are achieved. So we get the target calculi  $\lambda^K$  for this step and the following are features of the target calculi  $\lambda^K$ :

- Nearly linear (exception  $\text{if}\theta(v, e_1, e_2)$ )
- Only one abstraction mechanism ( $\text{fix}$ )
- Invoke continuation via “ $\rightarrow$ void”
- Termination instruction  $\text{halt}[\tau]v$

The static and operational semantics for  $\lambda^K$  are completely standard expect for the later two specialties.

The implementation of CPS-conversion, both type translation and term translation, is based on Harper and Lillibridge [1] and Danvy and Filinski [1]. The call-by-value translation proposed by Harper and Lillibridge is used to guide the type translations here. And the two-level system and the tail-recursion optimizations applied in the term

translation derive from Danvy and Filinski. So simultaneously CPS term conversion, tail-call optimization and elimination of static redices are accomplished.

Step 2:

$$\lambda^K \xrightarrow{\text{Closure conversion}} \lambda^C$$

The objective of this closure conversion step is to separate program code from data and this is achieved in two steps. The first step is closure conversion proper and the second step is hoisting, i.e., lift the closed code to the top of the program. Since most the work is done in the first step, the details of this step will be given.

The original idea of the way to typed closure conversion comes from Minamide *et al.*[3,5]. The approach Minamide *et al.* discussed desires a type-passing interpretation of polymorphism (Figure 2), which means types are constructed and passed to polymorphic functions at run-time. However type-passing interpretation has a lot of shortcomings, it requires both abstraction and translucent types which means duplication of constructs at the term and type levels. And abstraction is lost since there is no way to hold types abstract if they can always be analyzed. For our case most importantly it really complicates the closure conversion and makes it inefficient.

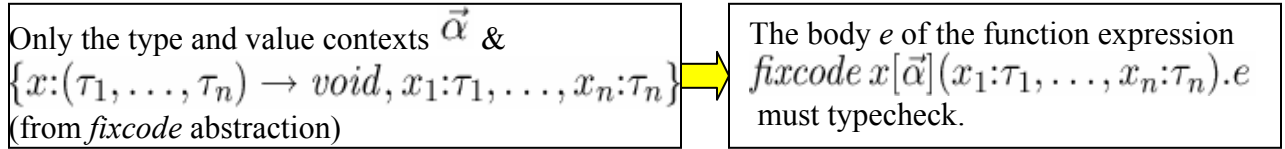
A type-erasure interpretation of polymorphism (Figure 3) as in *The Definition of Standard ML* [1,2] is proposed to solve problems with the type-passing interpretation. Terms representing types are passed instead of types themselves, i.e., run-time type information is represented by ordinary terms. And this makes the closure conversion much simpler and much more efficient. Now everything happening in the run-time is described by the terms. Neither abstract kinds nor translucent types are need as there is no type environment. Only type representations are passed when needed, and even traditional code optimizer can help. But this method also has its costs, such as subtle interactions with side-effects. This problem is taken care by “forcing polymorphic abstractions to be values”[1,2].

The key technical point of the implementation of the type-erasure method is the partial application of functions to type arguments to be values. What does this mean? Here is one example from which you can get the flavor:

$$\begin{array}{l}
 v \text{ has the type } \forall[\vec{\alpha}, \vec{\beta}].(\vec{\tau}) \rightarrow \text{void}, \\
 \text{type variables } \vec{\alpha} \text{ are intended for type environment} \\
 \text{type variables } \vec{\beta} \text{ are intended for the function's type arguments} \\
 \vec{\sigma} \text{ is a vector of types to be used for the type environment}
 \end{array}
 \left. \vphantom{\begin{array}{l} v \text{ has the type } \forall[\vec{\alpha}, \vec{\beta}].(\vec{\tau}) \rightarrow \text{void}, \\ \text{type variables } \vec{\alpha} \text{ are intended for type environment} \\ \text{type variables } \vec{\beta} \text{ are intended for the function's type arguments} \\ \vec{\sigma} \text{ is a vector of types to be used for the type environment} \end{array}} \right\}
 \begin{array}{l}
 v[\vec{\sigma}] \text{ is treated as values,} \\
 \text{and has type} \\
 \forall[\vec{\beta}].(\vec{\tau}[\vec{\sigma}/\vec{\alpha}]) \rightarrow \text{void}
 \end{array}$$

Except for this, the syntax of  $\lambda^C$  is similar to  $\lambda^K$ . As to the static semantics of  $\lambda^C$ , there is one thing to emphasize: **Code must be closed**

$$\frac{\vec{\alpha} \vdash_C \tau_i \quad \vec{\alpha}; \{x:\forall[\vec{\alpha}].(\vec{\tau}) \rightarrow void, x_1:\tau_1, \dots, x_n:\tau_n\} \vdash_C e}{\Delta; \Gamma \vdash_C \text{fixcode } x[\vec{\alpha}](x_1:\tau_1, \dots, x_n:\tau_n).e : \forall[\vec{\alpha}].(\vec{\tau}) \rightarrow void}$$

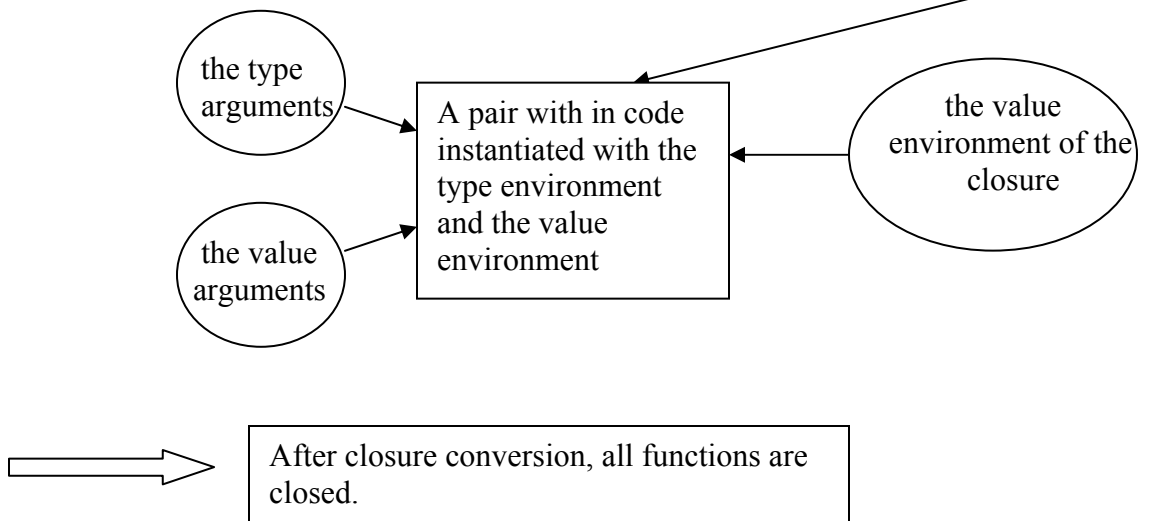


The closure conversion algorithm is constructed as a type-directed translation. The term translation is defined by the judgment  $\Delta; \Gamma \vdash_K e_{cps} \rightarrow e_{clos}$  and the value translation is formalized by judgment  $\Delta; \Gamma \vdash_K v_{cps} : \tau \rightarrow v_{clos}$ . Both of these two make sure what we get are correct closure conversion of terms and values from the CPS ones.

The key to translation is the type translation, which is defined as flowing:

$$\begin{aligned} & C[\forall[\vec{\alpha}].(\tau_1, \dots, \tau_k) \rightarrow void] \\ & \quad \underline{\text{def}} \\ & \exists \beta. \langle \forall[\vec{\alpha}].(\beta, C[\tau_1], \dots, C[\tau_k]) \rightarrow void, \beta \rangle \end{aligned}$$

The existentially-quantified variable  $\beta$  is the type of the value environment for the closure



The second step of closure conversion is hoisting. That is, closed function code is lifted to the top of the program, implementing the desired separation between code and data. This step is kind of elementary, and the changes to  $\lambda^C$  are limited.

- fixcode* is no longer a value
- Using *letrec* prefix to define code block at the top-level
- Code must be referred to by a new value form *labels(l)*

Step 3:

$\lambda^C \xrightarrow{\text{Allocation}} \lambda^A$

The objective of this step is to explicitly allocate spaces for tuples and fill them out field by field. The need for allocation generates from the fact that  $\lambda^C$  still form tuples via an atomic constructor.

Syntax of  $\lambda^A$  (similar to  $\lambda^C$ ):

<i>types</i>	$\tau ::= \alpha \mid \text{int} \mid \forall[\vec{\alpha}].(\vec{\tau}) \rightarrow \text{void} \mid \langle \tau_1^{\varphi_1}, \dots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha. \tau$
<i>initialization flags</i>	$\varphi ::= 0 \mid 1$
<i>terms</i>	$e ::= \text{let } \vec{d} \text{ in } v(\vec{v}) \mid \text{let } \vec{d} \text{ in } \text{if0}(v, e_1, e_2) \mid \text{let } \vec{d} \text{ in } \text{halt}[\tau]v$
<i>declarations</i>	$d ::= x = v \mid x = \pi_i(v) \mid x = v_1 \text{ p } v_2 \mid [\alpha, x] = \text{unpack } v \mid x = \text{malloc}[\vec{\tau}] \mid x = v[i] \leftarrow v'$
<i>values</i>	$v ::= x \mid \ell \mid i \mid v[\tau] \mid \text{pack } [\tau, v] \text{ as } \exists \alpha. \tau'$
<i>blocks</i>	$b ::= \ell = \text{code}[\vec{\alpha}](x_1:\tau_1, \dots, x_k:\tau_k).e$
<i>programs</i>	$P ::= \text{letrec } \vec{b} \text{ in } e$

We can see clearly that additional syntax, *initialization flags* and *declarations*, is consistent with the purpose of allocation. And tuple is no longer of a value form. The creation of an n-element tuple is achieved with two steps: an allocation step and n initialization steps. Again we will illustrate this using one example.

Ex: Creation of the pair  $\langle v_0, v_1 \rangle$

1.  $\text{let } x_0:\langle \text{int}^0, \text{int}^0 \rangle = \text{malloc}[\text{int}, \text{int}]$   
 -- allocates an uninitialized tuple and binds the address of the tuple to  $x_0$   
 --“0” superscriptes indicate the field is uninitialized
- 2.1.  $x_1:\langle \text{int}^1, \text{int}^0 \rangle = x_0[0] \leftarrow v_0$   
 -- updates the first field of the tuple with  $v_0$   
 -- binds the address of the tuple to  $x_1$
- 2.2.  $x:\langle \text{int}^1, \text{int}^1 \rangle = x_1[1] \leftarrow v_1$   
 -- initializes the second field of the tuple with  $v_1$   
 -- binds the address of the tuple to  $x$
- 
- 
- 

The initialization flag ensures that before a field is projected, it has to be initialized first, however initialization flag doesn't promise that a field is initialized only once. This is because  $x[i] \leftarrow v$  is interpreted as an imperative operation, and so at the end of the sequence  $x_0, x_1, x$  are all aliases for the same location. For the purpose of simplification, this problem is ignored for the project.

Then the last things are check type and term translation rules. They are all consistent with the allocation purpose. For type translation (from  $\lambda^C$  to  $\lambda^K$ ), this means the initialization flags are added to each field of the tuple types:

$$\mathcal{A}[\langle \tau_1, \dots, \tau_n \rangle] \stackrel{\text{def}}{=} \langle \mathcal{A}[\tau_1]^1, \dots, \mathcal{A}[\tau_n]^1 \rangle$$

And for term translation, declaration with *malloc* and initialization are considered for tuple values.

Step 4:

$\lambda^A$  Code generation TAL

So far the type structure we have in hand is composed by a combined abstraction mechanism, which abstracts a type environment, a set of type arguments and a set of value arguments, existential types, other data abstraction and initialized (or not) n-tuples. The major typing structure is almost built up in  $\lambda^A$ , so the objective of this stage is to generate code for TAL, most syntactic.

Before talking in detail about TAL, let me provide a dictionary of technical concepts and corresponding TAL syntax:

- Heap: a mapping of labels to heap values  

$$\underline{H} ::= \{ \ell_1 \mapsto h_1, \dots, \ell_n \mapsto h_n \}$$
  - Register file: a mapping of register to word values  

$$\underline{R} ::= \{ r_1 \mapsto w_1, \dots, r_n \mapsto w_n \}$$
  - A sequence of instructions  

$$\begin{aligned} \iota ::= & \text{add } r_d, r_s, v \mid \text{bnz } r, v \mid \text{ld } r_d, r_s[i] \mid \\ & \text{malloc } r_d[\tau] \mid \text{mov } r_d, v \mid \text{mul } r_d, r_s, v \mid \\ & \text{sto } r_d[i], r_s \mid \text{sub } r_d, r_s, v \mid \text{unpack } [\alpha, r_d], v \\ S ::= & \iota; S \mid \text{jmp } v \mid \text{halt}[\tau] \end{aligned}$$
  - Heap values: tuples and code  

$$h ::= \langle w_1, \dots, w_n \rangle \mid \text{code}[\vec{\alpha}]\Gamma.S$$
  - Word values: labels, integers, junk values ( $? \tau$ ), instantiation of word values and existential packages  

$$w ::= \ell \mid i \mid ? \tau \mid w[\tau] \mid \text{pack } [\tau, w] \text{ as } \tau'$$
  - Small value: word value, register or an instantiated or packed small value  

$$v ::= r \mid w \mid v[\tau] \mid \text{pack } [\tau, v] \text{ as } \tau'$$
  - Large value: tuples and code blocks – must be heap allocated
  - Code blocks: linear sequences of instructions
- } TAL program  
P ::= (H, R, S)

And there are a few technical points, which would help us with the understanding and development of TAL. TAL uses register names instead of alpha-varying variables used by  $\lambda^A$ , and this is the key difference between  $\lambda^A$  and TAL. Consequently a register calling convention must be used and made type-explicit in code generation. To illustrate this, again a small example,

E.g. To describe the code labels

TAL	$\lambda^A$ function type
$\forall[\vec{\alpha}]\{r1:\tau_1, \dots, rn:\tau_n\}$	$\forall[\vec{\alpha}].(\tau_1, \dots, \tau_n) \rightarrow \text{void.}$

Difference: for TAL, fixed registers are assigned to the arguments of the code  
 So if it's necessary to jump to a code block of this type, we need to make sure the type variables  $\alpha$  have been suitably instantiated and the registers  $r_1$  through  $r_n$  must contain values of type  $\tau_1$  through  $\tau_n$ . What's more, register must contain word values and types of TAL include type variables, integers, existentials and tuple types augmented with initialization flags.

TAL operational semantics is formalized as a deterministic rewriting system:

$$P \rightarrow P' \text{ (mapping program to program)}$$

Its well-formed terminal configuration have the form:  $(H, R\{\mathbf{r1} \mapsto w\}, \text{halt}[\tau])$  and this corresponds to a machine state where the computed value is contained by the register  $r_l$ , then all the other terminal configuration are all “stuck” program. Although on the Closure conversion stage, it will ultimately implement a type-erasure interpretation of polymorphism, here no type-erasure is suggested for the operational semantics, because by doing so, it will help the state and prove of the Subject reduction theorem, which helps to establish type safety for TAL. *If  $\vdash_{\text{TAL}} P$  and  $P \mapsto P'$  then  $\vdash_{\text{TAL}} P'$ .*

But when come to the instructions, we go back to the type-erasure approach. From the following chart, you can see the process is much simplified since here only two instructions we need to redefine their implementations.

One to one correspondence with Conventional assembly language	Two exceptions
<pre> <b>jmp</b> v <b>ld</b> r<sub>d</sub>, r<sub>s</sub>[i] <b>sto</b> r<sub>d</sub>[i], r<sub>s</sub> <b>bnz</b> r, v                     </pre>	<pre> <b>unpack</b> [α, r<sub>d</sub>], v <b>malloc</b> r<sub>d</sub>[τ<sub>1</sub>, ..., τ<sub>n</sub>]                     </pre>

Examples

TAL static semantics is derived from Morrisett and Harper’s [1]. It consists of 13 judgments.

Judgment	Meaning	
$\Delta \vdash_{\text{TAL}} \tau$ type	$\tau$ is a well-formed type	A5
$\vdash_{\text{TAL}} \Psi$ htype	$\Psi$ is a well-formed heap type	
$\Delta \vdash_{\text{TAL}} \Gamma$ rftype	$\Gamma$ is a well-formed register file type	
$\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2$ type	$\tau_1$ is a subtype of $\tau_2$	
$\Delta \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2$ rftype	$\Gamma_1$ is register file subtype of $\Gamma_2$	
$\vdash_{\text{TAL}} H : \Psi$ heap	$H$ is a well-formed heap of heap type $\Psi$	B2
$\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile	$R$ is a well-formed register file of register file type $\Gamma$	
$\Psi \vdash_{\text{TAL}} h : \tau$ hval	$h$ is a well-formed heap value of type $\tau$	
$\Psi; \Delta \vdash_{\text{TAL}} w : \tau$ wval	$w$ is a well-formed word value of type $\tau$	C4
$\Psi; \Delta \vdash_{\text{TAL}} w : \tau^\varphi$ fwval	$w$ is a well-formed word value of flagged type $\tau^\varphi$ (i.e., $w$ has type $\tau$ or $w$ is $?\tau$ and $\varphi$ is 0)	
$\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \tau$	$v$ is a well-formed small value of type $\tau$	D2
$\Psi; \Delta; \Gamma \vdash_{\text{TAL}} S$	$S$ is a well-formed instruction sequence	
$\vdash_{\text{TAL}} P$	$P$ is a well-formed program	

The first 5 (A5) judgments all state the well formedness and subtyping of various kinds of types. Except for the judgment for the heap types, which are in closed form along with heap, so no type context is used, other well-formedness judgments all are relevant to a type context. And corresponding to the structure of TAL program, the last two judgments (D2) assert well formedness of instruction sequences and programs. The 6<sup>th</sup> and 7<sup>th</sup> judgments (B2) assign types to heaps and to register files. Although no free type variables are considered for both of them, reference to heaps need to be contained in register file. Finally C4, the 8<sup>th</sup>, 9<sup>th</sup>, 10<sup>th</sup> and 11<sup>th</sup> judgments assign types to values. And need to note that the 11<sup>th</sup> rule is for assigning *flagged type* to word values:  $?\tau$  may be assigned the flagged type  $\tau^0$ , instead of any regular type.

Finally we come to the topic of code generation, which is the objective of this stage. Again we state this from both type translation and term translation. Type translation is straightforward, except  $\forall$  for which registers must be assigned to value arguments. The term translation is also very straightforward. However, registers must be kept track with, including both mapped and used registers, since then fresh register can be assigned. And translation handles fresh label generation explicitly by supplying a set of used labels to the term and block translation judgments [2]. The rest things in this translation, are just translating values to small values, terms to pairs of instruction sequences and heaps, blocks to heaps and programs to programs.

Before the resulting TAL code may be run on a real machine, a compiler must perform a final register allocation step to ensure that the TAL code runs with the number of registers supplied by the machine. This is viewed as a type-preserving optimization within the framework of TAL. Either a clever register or a practical register is implementable as a type-preserving transformation on TAL code [2].

### Summary

Based on a conventional RISC assembly language, type-correct source languages are mapped to type-correct assembly language and the translation from system F to TAL is a sequence of type-preserving transformations. So the compiler for the whole transformation  $\vdash_F e: \tau \rightarrow P$ : is a type-preserving compiler. It takes well-typed  $\lambda^F$  terms and returns well-typed TAL code. The polymorphic closure conversion applied in the 2<sup>nd</sup> stage is viewed as a breakthrough in this field. TAL provides a low-level, statically typed target language and it's suitable to support a wide variety of source languages. Its type-check ability and *proof carrying code* feature will make the implementation of its application and practice more promising.

### References

- [1] Greg Morrisett, David Walker, Karl Cray, and Neal Glew **From System F to Typed Assembly Language** In the *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85-97, San Diego, CA, USA, January 1998.
- [2] Greg Morrisett, David Walker, Karl Cray, and Neal Glew **From System F to Typed Assembly Language (Extended version). Technical Report TR97-1651**, Cornell University, November 1997

- [3] Stephanie Weirich , Karl Crary and Greg Morrisett **Intensional Polymorphism in Type-Erasure Semantics** *1998 International Conference on Functional Programming*, pages 301-312, Baltimore, September 1998.
- [4] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic **TALx86: A Realistic Typed Assembly Language** In the *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25-35, Atlanta, GA, USA, May 1999
- [5] Stephanie Weirich **Intensional Polymorphism in Type-Erasure Semantics International Conference Presentation**, 1998