Typed Intermediate Languages Inside and Outside of Compilers

ÁNOQ of the Sun, Hardcore Processing *

May 3, 2000

1 Abstract

This document is for a talk about the use of intermediate languages in compilers - and even outside of compilers. The talk aims to give an overview of the projects FLINT, MLRISC, TAL and TILT which are all different kinds of intermediate langauges. Intermediate languages in compilers are usually meant to separate different phases of a compiler thus making the compiler more modular. This also makes it possible to support many different target languages and source languages in the compiler while sharing for instance the optimizing parts of the compiler. Some modern intermediate langauges are typed which usually means that programs can be verified to be correct by doing a simple typecheck on the intermediate form of the program. This makes typed intermediate languages suitable as a format for distribution of software components. There are also other interesting applications of these languages which will be mentioned in the talk.

2 MLRISC

MLRISC is a compiler backend which can translate from a relatively lowlevel typed intermediate language to multiple target machine languages - mostly for general purpose CPUs.

2.1 Goals of MLRISC

- To make it possible to translate into and optimize for multiple different target languages - whether it is very well designed RISC processors or heavily entwined processors like the x86 family of processors.
- To supply an intermediate language suitable for use with many different source languages - ranging from (but not limited to) imperative and object oriented langauges to advanced functional languages like ML and Haskell.
- To enable very advanced program optimizations.
- To be able to reuse those optimizations both for different source languages and target languages.

2.2 Supported targets

- Dec Alpha
- HPPA
- x86
- Sparc
- PowerPC
- TI C6x (Texas Instrument's C6x series of VLIW-like DSP)

A MIPS backend is not complete yet, however more and more target specific parts are being generated from a machine description.

2.3 MLRISC is being used for real

MLRISC is already being used in the following projects:

- SML/NJ - one of the popular Standard ML compilers
- TIL - described later
- Tiger - the language used in Andrew Appel's book "Modern Compiler Implementation in "(ML|Java|C)
- C– - a C-like assembler language
- Moby - a test-bed language for the design of ML'2000
- CHILL - a telecommunications language
- The Church project

2.4 How does it work

- MLRISC is customized according to the needs of both source language and target architecture.
- The compiler using MLRISC generates MLRISC instructions in the *specialized* intermediate language - i.e. as the intermediate language looks after MLRISC has been customized appropriately. Notice that MLRISC assumes that all target independent optimizations has already been made.
- A function *codegen* is built by assembling customized optimization modules. Calling the *codegen* function will compile programs from the customized MLTREE language to the target architecture.
- MLRISC performs only few optimizations directly on the MLRISC intermediate language.
- Performs most of the optimization on the actual target machine language to take advantage of each architecture.

2.5 More about the MLTREE language

- The MLRISC intermediate langauge is defined as ML datatypes called the MLTREE language. MLTREE has an infinite number of registers but with the first 0..K-1 registers mapping directly to special registers of the target machine.
- The MLTREE language can be customized with regard to constants, regions, pseudo-ops, annotations and user defined extensions.

2.6 Other things

- MLRISC has a generator called *MDGen* which can generate customized modules from a machine descriptions. Machine discriptions for the supported backends are readily available.
- There is a garbage collection safety framework. Seems like a good thing.
- Graphical interface to control flow graphics etc. of generated code.

2.7 Future work

- Detailed user manuals :) A nice tutorial to get started will soon be available.
- Support for GC (seems to be almost there).
- Currently, the framework for predicated VLIW architectures compilation is incomplete, and contain only one back end (C6)
- Ideas from typed assembly language and proff carrying code is to be investigated for the project.

3 Typed Assembly Language (TAL)

- TAL is an extension of traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules.
- So, instead of being a typed intermediate language, TAL is actually close to being a typed *target* language - indeed the TAL compiler will compile the TAL code into a typed target language.
- It is currently only implemented for Intel's IA32 architecture where it is called TALx86.
- The goal of the TAL project is to extend the paradigm of type-directed compilation to its limit.

3.1 What does it do?

- The typing rules guarantee memory safety, control flow safety, and type safety of TAL programs. As a result, TAL code is a form of proof-carrying code.
- The typing constructs are expressive enough to encode most source language programming features including records and structures, arrays, higher-order and polymorphic functions, exceptions, abstract data types, subtyping, and modules.
- TAL is flexible enough to admit many low-level compiler optimizations.

3.2 Current uses

- The compiler TALC compiles TALx86 into code for the Intel 32-bit family of assembly languages. This assembly code is annotated with type information that can be verified by our type checker before the code is assembled by the MASM assembler.
- A prototype compiler called *Popcorn* for a safe C-like language which generates TALx86 code has been implemented.
- Another prototype compiler called *SCHEME–* has been written which generates TALx86 code. This compiler is written in the *Popcorn* language.

3.3 Potentials and capabilities

- TAL is an ideal target platform for type-directed compilers that want to produce verifiably safe code for use in secure mobile code applications or extensible operating system kernels.
- Compiler writers can use the safety of TAL programs to debug sophisticated program transformations such as closure conversion and optimizations like datatype specialization.
- The types helps to both check the correctness of transformations and to enable analyses or optimizations that are extremely difficult without types.
- TAL applets, like Java applets, may be downloaded from untrusted sources on the internet, verified, and executed safely without fear they will corrupt the host machine. Furthermore, because TAL is assembly language, there is no interpretation overhead during this process and no just-in-time compiler to run or trust.
- The present TAL effort many applications in the security and compilation domains.

3.4 Current project status

The software is currently pre-alpha, contains little documentation, and comes with absolutely no warranty. Use at your own risk! :)

4 Typed Intermediate Language Two (TILT)

- The goal of the project is to produce a compiler for the ML-family of programming languages (SML'97, Caml Special Light, and KML) that takes advantage of types throughout compilation in order to produce better code without compromising safety or correctness.
- Compilation is done by always manipulating strongly typed intermediate languages so that the intermediate programs always can be typechecked.

4.1 Reasons to typecheck

- Detect and isolate bugs in the compiler.
- Take advantage of types to produce smaller, faster code.
- Send low-level, heavily optimized code over the network to untrusting hosts.
- Construct proofs of compiler correctness.

4.2 About the first TIL compiler

- On DEC Alpha it compiles programs with 3 times the performance and one fifth of the total heap garbage collection compared to SML/NJ (in 1996).
- However compilation time is (was) 8 times slower and it does not have full support for modules.
- The compiler features nearly tag-free data - meaning that it only uses tags on heapallocated data and not on data in registers or on the stack.
- TIL also features highly unboxed datastructures.
- However it is slower for some polymorhic functions which requires typecasing and runtime manipulation of typeinformation.
- TIL uses intensional polymorphism to achieve much of it's magic.
- The compiler generally does a lot of optimizations - all of which is done on the typed intermediate languages of the compiler.

4.3 Project status

The sources for the original TIL compiler are available for perusal but it is unsupported. It seems that there is no notion of a current version of the TILT compiler - however I have not tried asking for it.

5 FLINT

- The intentions are that FLINT can serve as a common intermediate format for many advanced type safe languages.
- FLINT proposes to fully realize the ultimate impact of Proof Carrying Code (PCC) technology by scaling it up to real programming languages, production compilers, and sophisticated security policies.
- Particular attention will be given to scalability, interoperability, efficiency, and the principled interaction of security policies with containment mechanisms.
- A goal of the project is language independence, meaning that by relying on general axiom sets code producers have a lot of flexibility in choice of programming languages and compilers.
- The project intends provide a high-quality, robust, optimizing, certifying compiler for ML and for Java.

5.1 The FLINT language

- It is a kind of explicitly typed lambda calculus containing constructors.
- It it intended to be used as middle ends in compilers, meaning that it is to be put after parsing, type-checking and pattern-match compilation but before the backend which is doing loop optimizations, CPS-based contractions, closure conversion and machine code generation.
- It has simple and well-defined semantics.
- It has expressiveness, to support advanced features such as closures, ML-like polymorphism and modules, and Java-like classes and objects.
- Pay-as-you-go efficiency, so that programs that do not use a particular feature should not be implemented less efficiently simply because they may interact with those that do use the feature.
- Suitable for standard program optimization.
- Support for system programming: Contains primitives such as bit-manipulation, safe type-cast, dynamic types, safe marshalling, arbitrary checkpointing, and real-time memory management.
- FLINT is designed to be extensibile.
- FLINT itself is doing simple dataflow optimizations and local or cross-module type specializations before it produces an optimized version of the FLINT code.

5.2 Some comparison with Java bytecode

Currently, Java Virtual Machine Language (JVML or Java bytecode) is an important langauge in the area of platform independence, so here are a few comparisons to FLINT:

- JVML is Java-specific; FLINT is language-independent, allowing Java and FLINT applets to coexist (since JVML can be compiled into FLINT!).
- Unlike JVML, FLINT has a clean and well-founded semantics so it is more trustworthy.
- Cutting-edge research on PCC and secure information flow can be more easily incorporated into FLINT.

On a side note - the security model that is now used by the Sun, Netscape, and Microsoft Java implementations was first explained and justified by the research of the authors of FLINT :)

5.3 Features and plans

- Type directed compilation is carried over across the higher-order module boundaries, even for higher-order functors.
- Recursive and mutable data objects can use unboxed representations without incurring expensive runtime cost on heavily polymorphic code. This is where TIL uses a type-passing approach which incurs heavy-weight runtime type analysis and code manipulation.
- Parametrized modules (functors) can be selectively specialized, just as normal polymorphic functions.
- Special type representations are used to reduce the cost of type manipulation and thus the compilation time.
- Adding new type constructors to FLINT should not give any type reconstruction problem.
- In the long term, extensions like the following are planned: n-bit reals and integers, C-like records (structs) and Haskell-like Thunks.

5.4 Project status

- FLINT is already being used in the SML/NJ compiler and in another compiler called FLINT/ML.
- FLINT should currently be undergoing a major redesign.

References

[1] Zhong Shao "An Overview of the FLINT/ML Compiler", Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97).

[2] Andrew W. Appel, Edward W. Felten, Zhong Shao "Scaling Proof-Carrying Code to Production Compilers and Security Policies ", January 1999 on the FLINT website.

[3] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee "TIL: A Type-Directed Optimizing Compiler for ML", 1996 SIGPLAN Conference on Programming Language Design and Implementation.

[4] Lal George, Allen Leung "The MLRISC website": `http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/index.html`.

[5] "The TAL website" : `http://www.cs.cornell.edu/talc/`

[6] "The TILT website" : `http://www.cs.cornell.edu/Info/People/jgm/tilt.html`

[7] "The FLINT website" : `http://flint.cs.yale.edu/`