A Compile-Time Infrastructure for GCC Using Haskell

Peter Collingbourne and Paul H J Kelly

Department of Computing, Imperial College London, London SW7 2AZ, UK
{peter.collingbourne,p.kelly}@imperial.ac.uk

Abstract. This project aims to improve the metaprogramming and verification capabilities of the GNU Compiler Collection. It consists of a plugin infrastructure that exposes two compile-time mechanisms. The first of these is a mechanism by which one may perform arbitrary computations over types within the context of the C++ template metaprogramming infrastructure. The second of these exposes selected portions of the control flow graph and basic block statements of Low GIMPLE, with additional support infrastructure that allows for data-flow analysis of the resultant structure. The plugins themselves are written in Haskell, reflecting the functional nature of both C++ template metaprogramming and standard data-flow analysis. We demonstrate the effectiveness of our technique using specific case studies in the field of session types.

Key words: C++, Metaprogramming, Program Analysis

1 Introduction

In the field of programming languages, compile-time computation is a technique for performance optimisation, software verification and program generation. Compile-time computation may also be used to extend a language, whereby the compile-time code "lowers" the extended language to a standard representation. Experienced LISP and Haskell programmers will be aware of relevant technology within their respective languages: LISP macros and Template Haskell.

The C++ language provides the somewhat limited template metaprogramming technique for compile-time computation. Whereas LISP macros and Template Haskell operate over executable code, template metaprogramming operates over types. Template metaprogramming was largely an unintended consequence of the language design. It provides a limited form of access to the type structure of a program, and does not provide access to the actual program code. It is also sometimes criticised for being rather obtuse and unmaintainable. Much of this criticism stems from the syntax, which can appear unintuitive to those not familiar with the C++ standard. Furthermore, there exist certain computations which are impractical to express using template metaprogramming, for which an easier alternative is desired. We shall discuss one such example in Section 5.

A number of approaches have been considered for introducing a more powerful form of compile-time computation to C++. Examples of approaches taken

include meta-object protocols, and databases with associated query languages. This project represents an alternative approach to improving the compile-time computation capabilities of the C++ language. This shall be accomplished from two angles, with particular view to high-level expressiveness. In particular, we attempt to provide programmatic, meta-level access to two aspects of the workings of the GNU Compiler Collection. The first of these is part of the C++ front-end, specifically the part of the template meta-programming infrastructure in which types are computed. The second is the mid-end of the compiler, in which statements, in a format known as Low GIMPLE, are processed, and control flow information is also available to us. As the midend is language-neutral, we expose the midend for every language supported by GCC. We utilise a plugin architecture for maximum flexibility. We use Haskell as the plugin implementation language. Each plugin consists of a Haskell function with a specified type, and the information contained within the GCC internal data structures are exposed to the plugin as Haskell data structures provided as function parameters. An attempt was made to design many of the Haskell data structures in a compiler-independent manner, or at least provide a compiler-independent access mechanism, in order that our extension may be ported to another compiler with minimal plugin changes necessary.

The two aspects of our extension have been designed together so that they may cooperate in a user-defined way using the C++ type system. For example, a type may be produced by the template metaprogramming extension with specific annotations which indicate that a certain control-flow analysis must be carried out to verify the usage of those variables with that type.

We claim the following contributions:

- The specification and implementation of a functional extension to the C++ language that extends the template metaprogramming mechanism.
- The specification and implementation of an augmentation of a compiler midend that allows for functional program analysis.
- An implementation of the session type communication protocol for C++ that provides advanced verification capabilities and presents a cleaner interface to the user using the above two extensions.

Our work exists only as a prototype and is emphatically not a language proposal. This work is an exploratory project to investigate language extensions that improve certain metaprogramming aspects of C++.

1.1 Motivation

Our extension can be utilised to support verification of generic and domain-specific constraints imposed on a language. An example of a generic constraint would be to prevent detectable dereferencing of null pointers. To verify this constraint, an analysis can be inserted as a midend pass. An example of a domain-specific constraint relates to domain-specific languages embedded within a program, such as SQL or regular expressions. A type system extension for regular

expressions can perform the dual role of verifying the validity of a regular expression, and compiling it in order to avoid runtime parsing. A midend pass targeted at SQL queries can verify the validity of a query's syntax and determine whether the input and output parameters to the query are of the correct type.

Our extension may also be used to create simpler template implementations. Let us now provide a small example in the context of session types, to be introduced later. Briefly, a session type represents a sequence of communication actions carried by a channel. There exists a specific type transformation over session types known as the *dual transformation*. The dual transformation is the substitution of all input operations within a session with output operations, and vice versa. We have developed a C++ implementation of session types [1] which represents a session type as a hierarchy of template instantiations; e.g. the type

seq<in<int>, seq<out<int>, end>>

represents a session consisting of an `int` input operation followed by an `int` output operation. Now consider a template function to implement the dual transformation. Using standard template metaprogramming, we must handle separately each particular template instance that may appear in the code, and propagate the function call accordingly. The resultant implementation consists of 53 lines of code. When implemented using Haskell, we may take advantage of two techniques in order to improve the conciseness of the code. The first is Haskell's *scrap your boilerplate* [2] design pattern. This design pattern allows one to write functions that traverse a data structure and operate on values of particular types without the need to implement a traversal manually, in a manner that is independent of the data structure being traversed. The second is a mechanism which translates shorthand type specifications written in a C++-like form into our full type specification. Our final implementation consists of six lines of code:

```
dual (CTStrP"in<?t>"[CP_t t]) = CTStrP"out<?t>"[CP_t t]
dual (CTStrP"out<?t>"[CP_t t]) = CTStrP"in<?t>"[CP_t t]
dual (CTStrP"call<?t>"[CP_t t]) = CTStrP"dual_call<?t>"[CP_t t]
dual (CTStrP"dual_call<?t>"[CP_t t]) = CTStrP"call<?t>"[CP_t t]
dual t = t
dualSessionType t = return (everywhere (mkT dual) t)
```

2 Background and Related Work

This work may immediately be contrasted with works such as LISP macros [3] and Template Haskell [4]. Both mechanisms expose a tree-based representation of the language to computation within that programming language. The key distinction between this work and ours is that in both of these systems, meta-computation occurs over an IR that is the program's AST. LISP and Haskell both pose challenges for analysis writers due to the difficulty of extracting control flow graphs from these. Furthermore, Template Haskell provides no easy way to extract the type of a given meta-expression. In our system, meta-computation occurs over a CFG representation of the IR, and type information is available.

Another frame of comparison is the meta-object protocol. Meta-object protocols provide a meta-level object-oriented interface to a programming language. They also provide the mechanism by which one may extend the syntax of a language (for example by adding a new domain-specific loop statement). These meta-level objects represent parts of the language, such as the compiler's intermediate representation and its mechanism for interpreting certain operations (such as adding two objects together).

Two meta-object protocols for C++ are Ishikawa et al's [5] MPC++ and Chiba's [6] OpenC++. Both MOPs expose the meta-level architecture using a C++ model, and allow the user to create new constructs simply by extending the relevant classes of the C++ representation.

Existing C++ MOPs can be distinguished from our work in two ways. Firstly, no C++ MOP that we are aware of provides access to the template instantiation mechanism in the same way as put forth in this work. Secondly, as with the above discussed macro systems, these MOPs provide access to a high-level IR similar to an AST, and no control flow graph is made available.

We may also consider mechanisms for exposing intermediate representations as databases with associated query languages. Examples include a Prolog-based system known as Deepweaver [7] and a Datalog-based system [8]. These systems make the task of discovering the control-flow simpler than the C++ MOPs we have discussed. For example, the Datalog-based system provides access to the control-flow graph, and Deepweaver provides a reaching definition analysis. However both systems presently lack a generalised data-flow analysis framework.

Works that attempt to provide generic data-flow analysis to C++ via a compiler include gdfa: A Generic Data Flow Analyzer for GCC [9]. This is a generic data-flow analysis extension to GCC, implemented in C. Currently, the type of data flow information is defined by the framework to be a bit vector. The authors plan a more flexible approach which permits any type of data flow information. Another such work is the generic data-flow analysis component of ROSE [10], a source-to-source program transformation and analysis framework, in which an arbitrary data type can be used for data flow information.

The plugin-based MELT [11] project exposes the GCC midend to a higher-level language. The language is domain-specific and is a LISP variant. The authors have opted for a compiler-specific approach (for example, plugins have direct access to GCC trees). Compilation is achieved via translation to C; MELT imports GCC functions and macros via the definition of translation rules. The goal is an abstract interpretation framework for GCC, which as of the time of writing appears to be unimplemented, however the potential exists for a generic, compiler-independent analysis framework to be implemented.

3 Template Metaprogramming Extension

3.1 Design

In order to use the template metaprogramming extension, the user provides a *transformation function* written in Haskell, whose role it is to transform a C++

type in some specified way, and annotates a `typedef` statement with a custom attribute containing enough information to allow the compiler to locate the implementation of the transformation function. Under our type system extension, if the `typedef` is annotated with such an attribute, the source type (the type appearing on the left-hand side of the `typedef` statement) will be passed to the transformation function. The result of the transformation function will then receive the alias of the target name (the name appearing on the right-hand side of the statement) in the symbol table.

Transformation functions are provided with a single argument: a Haskell representation of a C++ type (a value of type `CxxType`). `CxxType` is an *algebraic data type*; briefly, a data type built from a number of constructors. Each constructor has parameter types associated with it, and may be used to build a value of that type by supplying values of the parameter types as arguments to the constructor. `CxxType` is constructed from a number of ADTs that reflect the structure of the C++ type system. For example, the type that represents a template argument, `CxxArgument`, has three constructors as shown below.

```
data CxxArgument = CAType CxxType
                 | CAValue CxxValue
                 | CATemplate CxxTemplate
```

This reflects the fact that a template argument in C++ may be a type, value or template. The remainder of the type specification is given in the extended version of this paper [12].

The transformation function returns a value of type `CxxCompilerM CxxType`, where `CxxCompilerM` is a wrapper around the `IO` monad. Wrapping the `IO` monad was done for the sake of safety by preventing the transformation function from accessing the underlying monad and performing unsafe I/O actions; as well as portability since the monad may be substituted by another should this be required for a particular compiler implementation.

We provide a number of functions that allow type information contained within the compiler to be accessed. These functions are designed to mirror the capabilities of standard C++ metaprogramming. The type signatures of the two most important functions are shown below.

```
getMember :: CxxNamedType->String->CxxCompilerM (Maybe CxxType)
buildStruct :: Maybe String
            -> (CxxNamedType->CxxCompilerM [(String,CxxType)])
            -> CxxCompilerM CxxNamedType
```

`getMember` looks up and returns the field with the given name within the given type, and is equivalent to use of the `::` operator in standard template metaprogramming. `buildStruct` defines a composite type with the given `typedef` fields and optional name and returns the name of the constructed type, and is equivalent to defining a parameterised inner class. It is designed to allow for the definition of composite types that refer to themselves, i.e. recursive types, as well as corecursive (mutually dependent) types. As we shall see in Section 5.2, this capability has applications in the context of session types.

To improve conciseness, one may use a shorthand representation of a C++ type as an alternative to the long-winded full representation. This representation consists of a C++ type expressed in the standard way, but with "holes" into which one may substitute values of a variety of types, in a similar fashion to the C `printf` and `scanf` functions. For example, the shorthand representation of an instance of a template `in` with a single argument t is CTStrP "in<?t>"[CP_t t], whereas the full representation contains 91 characters.

3.2 Implementation

An implementation of the language extension is responsible for recognising our custom attribute and invoking the desired transformation function contained within the Haskell plugin with the appropriate parameters. In order to achieve this, it must be capable of converting the compiler's internal representation of the type to our Haskell representation, and vice versa. It must also provide an implementation of the compiler interface functions discussed in Section 3.1.

Our implementation uses the development version of GCC as a baseline, and tracks developments to the mainline branch. Our extension is currently based on revision 142782 of the GCC Subversion repository. The Haskell compiler used is GHC 6.8.2. In order to prepare a transformation function for use by our language extension, it must first be placed in a Haskell source file and compiled using GHC. The custom attribute contained in the relevant `typedef` thus supplies the path to the Haskell object file containing the compiled function, and the (symbol) name of the function within the object file.

The implementation is written in Haskell (by necessity, as Haskell does not provide a mechanism for C programs to build Haskell data structures), and consequently it must have access to a number of GCC functions and macros. Access is provided via Haskell's FFI (foreign function interface) mechanism, using which Haskell may import arbitrary C functions. Macros are wrapped in a function by necessity; we also wrap functions for the sake of safety, as the compiler will flag an error if the signatures of any of the functions changes, whereas this will not happen if they are imported directly.

We modified the attribute-handling code for the C++ front end to recognise our attribute and annotate the relevant `TYPE_DECL` tree node with this information. A hook was inserted into the routine responsible for performing template substitutions over declarations to recognise this annotation and call the Haskell code responsible for loading and invoking the transformation function.

We must ensure that our extension can coexist with GCC's garbage collector (GGC). We believe, but have not formally verified, that our extension preserves soundness and completeness with respect to garbage collection. Space restrictions preclude us from elaborating; the extended paper [12] contains details.

4 Midend Passes in Haskell

In a similar fashion to the metaprogramming extension, a Haskell midend pass is implemented as a transformation function. GCC midend passes are invoked

once for each function defined within the current translation unit. Thus the input parameter to the transformation function is a representation of the function undergoing analysis. A function's representation consists of its control flow graph, the node IDs corresponding to the entry and exit nodes and a mapping between variables referenced in the CFG and their types. The CFG is represented using Haskell's inductive graph [13] representation. Each node contains a list of statements that make up that basic block.

The function returns a function representation of the same type as the input parameter. In the future, this return value will be converted back to the compiler internal representation thus permitting program transformation. The `CxxCompilerM` monad is also used here so that the analysis function may obtain type information, although it is prevented from creating new types.

We depart by necessity from our convention of compiler independence, as the precise representation of a statement will differ between compilers. Consequently, the basic block, statement and edge data types are implementation-defined. We have implemented `use` and `def` functions, generic accessor functions which respectively provide a list of variables used and defined within a statement, and plan to implement further accessor functions.

We have developed a generic implementation of the standard data-flow analysis. The design of the framework allows for intuitive, "textbook" analysis descriptions. For example, below is an implementation of the live variable analysis:

```
liveVariableAnalysis = DataFlowAnalysis
-- direction          xfer  boundary   meet         init
   BackwardAnalysis  xfer  Set.empty  Set.unions  Set.empty
  where
    xfer stmt lv = Set.fromList (use stmt) `Set.union`
      (lv `Set.difference` Set.fromList (def stmt))
```

The GCC implementation of this extension exposes selected portions of the Low GIMPLE intermediate representation as an algebraic data type. The Haskell transformation pass has been placed after the SSA transformation pass in order that ϕ nodes are available. This has the effect of simplifying certain analyses.

As with the metaprogramming extension, this extension is implemented in Haskell, and uses FFI to import any necessary functions and macros.

In order to use a midend pass plugin, an argument `-fhaskell=`*object*`,`*symbol* is provided to the GCC executable, where *object* is the name of the object file and *symbol* is the (symbol) name of the transformation function.

Because the type extraction functions are language-specific, we utilise GCC's langhooks mechanism to pass them from the frontend to our midend pass.

5 Case Study: Session Types

The examples revealed in this section show how we may use our extensions to implement library features which may be impossible to implement using standard C++. In particular, our examples shall concern *session types* [14], a means of

characterising dyadic interaction between processes over a communication channel. Process interactions are expressed as a sequence of communication actions, and any communication taking place over the session with which the type is associated must conform to the sequence of actions. By imposing a compile-time type constraint on a communication channel, we eliminate the possibility of communication type errors occurring at runtime, thereby improving code quality.

Session types are usually expressed as terms with a specified syntax, an instance of which we shall briefly describe. A session type may be an *action*. An action specifies a communication direction (in or out) and type (usually a primitive type). An action represents the reception or transmission of a value of the specified type. A session type may also be the *sequential composition* of two or more session types. The session type that is the composition of one or more session s_1, $s_2 ... s_n$ represents the actions in s_1, followed sequentially by those in s_2 and so on up to s_n. A session type may also be a *choice* between a number of sessions s_1, $s_2 ... s_n$. A session type may also be the *terminating session type*. A session with the terminating session type may only close the channel.

Session types may also be recursive, in that they may refer back to themselves in order to create infinite sequences of actions. The two standard [15] syntax elements used in recursion are $\mu t.s$ where s is a session, which defines a new recursive type t which refers to s, and t, where t has been defined in an enclosing μ type, which is equivalent to the entire corresponding μ.

As mentioned in Section 1.1 of this paper, we have an implementation of session types in C++ in which we represent a session type as a hierarchy of template instantiations. Actions in our implementation take the form **in**<T> or **out**<T>, where T is the primitive data type to be sent or received across the channel. As for sessions, sequential composition composes an action A with a continuation session S, and thus is represented as **seq**<A,S>. For the choice construct we compose sessions via a representation of the form **choice**<S1,S2,...,Sn>.

We also have a representation of recursive types in our implementation. As a type may not directly be defined in terms of itself, we have specified a three-stage protocol that may be used to define a recursive type. Firstly, an incomplete composite type s is defined. Secondly, the recursive session type r is defined using a `typedef`. Wherever a recursive reference is required, the special instantiation **call**<s> is used. Thirdly, s is fully defined, with an internal typedef t that is defined to be r. An example of such a definition is shown below.

```
struct s; typedef seq<in<int>,call<s> > r;
struct s { typedef r t; };
```

5.1 Subtyping

Our first example relates to the *subtyping* relationship between two session types. As with object-oriented programming, subtyping defines substitutability. Subtyping may also be used to determine compatibility between two peers with their own session types. The formal definition of the concept is shown below, where

idom and odom are respectively the input and output domains of a session, i.e. the set of types which may be immediately received or sent by the session.

Definition 1. *Type simulation [16, 1]. A type simulation is a relation R that satisfies the following property.*

$$\begin{aligned}(S_1, S_2) \in R \Rightarrow\ & \mathsf{idom}(S_1) \subseteq \mathsf{idom}(S_2) \\ & \wedge\ \mathsf{odom}(S_1) \supseteq \mathsf{odom}(S_2) \\ & \wedge\ \forall t \in \mathsf{idom}(S_1) \exists S_1', S_2' : (S_1 \xrightarrow{\texttt{in}\ t} S_1' \wedge S_2 \xrightarrow{\texttt{in}\ t} S_2' \wedge (S_1', S_2') \in R) \\ & \wedge\ \forall t \in \mathsf{odom}(S_2) \exists S_1', S_2' : (S_1 \xrightarrow{\texttt{out}\ t} S_1' \wedge S_2 \xrightarrow{\texttt{out}\ t} S_2' \wedge (S_1', S_2') \in R)\end{aligned}$$

Definition 2. *Subtyping. $S_1 \leq S_2$ iff there exists a type simulation R such that $(S_1, S_2) \in R$.*

We have developed a plugin which implements a template that computes a subtyping judgement about two session types. Firstly, the session types provided as parameters are converted to a Haskell map-based representation. Recursive references of the form call<s> are converted to monadic values which represent the retrieval of the appropriate session type from the compiler using `getMember`. Secondly, we make a subtyping judgement by attempting to build a type simulation as defined in Definition 1. The result is then returned to C++ as a type: `closure_true` if a type simulation was built, or `closure_false` if not.

At present, the subtyping judgement is used to define session substitutability. The custom type conversion constructor for the session class invokes a static assertion to verify that the type returned by our template was `closure_true`.

5.2 Session Type Parsing

Our second example concerns the parsing of textual representations of session types into the template instantiation hierarchy we have defined. For instance, the type parse<"?int.end">::t shall be an alias for **seq<in<int>, end> >**. Note the string parameter to `parse`; this represents a small extension to GCC's C++ front-end to allow it to accept string constants as template parameters. A generated LALR(1) parser is used to parse the string into a `CxxType`.

Our transformation function is also capable of defining recursive session types, using the three-stage protocol defined above. Where a session type of the form $\mu t.s$ (written `%t.s`) is encountered, a new composite type is created using `buildStruct` and the type s is processed within the function passed to `buildStruct`, which has access to the name of the type. We can thus create a mapping between recursive references and their types, which is used to insert the appropriate instance of `call` wherever a recursive reference is encountered.

5.3 Linearity Constraint

C++ is a statically typed language. However, session type theory [14] states that after a session has performed an action, its type mutates to its continuation type

relative to that action. Since C++ does not permit a variable's type to mutate, in our implementation we introduce a new session variable after each action.

After we have used a session variable (i.e. by sending or receiving over it), it becomes invalid. This means that any further use of the variable is an error and would violate our typing system. A variable with such a constraint imposed upon it is known as [17] a *linear variable*, and any program that satisfies this property is said to satisfy the *linearity constraint*.

We have implemented a midend verification pass that verifies the linearity constraint. It uses a backward data flow analysis that counts the number of times a variable is used before it is redefined (to ensure the monotonicity of the analysis a maximum of 2 is imposed on the usage count). Should any variable's usage count be more than 1 at any point, we use the variable type mapping to look up the type of that variable and determine whether the type was declared as linear, by checking for the presence of a particular `typedef` within the type (as a consequence, only composite types may be declared as linear at present).

5.4 Example

The below code shows the three plugins described above in operation.

```
1 typedef parse<"%s.?int.!int.?int.!int.s">::t st1;
2 typedef parse<"%s.?int.!int.s">::t st2;
3 typedef parse<"!int.%s.?int.!int.s">::t st2p;
4 void ch1(session<st1> s) {
5   session<st2> s1 = s;
6   session<st2p> s2; int x;
7   s2 = s1.read(x);
8   s2 = s1.read(x);
9 }
```

Lines 1–3 use the session type parsing mechanism to produce a session type in the required format. For example, the declaration of the type `st2` is equivalent to the following, modulo the name of the `struct`:

```
struct haskell_123;
typedef seq<in<int>,seq<out<int>,call<haskell_123>>> st2;
struct haskell_123 { typedef st2 t; };
```

Line 5 is a demonstration of a successful subtyping judgement; a cycle of two `int` inputs and outputs was deemed to be a subtype of a cycle of one.

Because the channel is read twice on lines 7 and 8, the code is invalid. The linearity checker will detect a violation: the linear variable `s1` is read twice without a redefinition. In the data flow analysis, this is represented as a usage count of 2 for this variable between the output of line 5 and the input of line 7.

6 Evaluation

Design A key design choice of this project was whether to implement our metaprogramming extension within a compiler frontend, or as an external

preprocessor. From a technical point of view, a preprocessor would be the preferred option as this would permit compiler independence. It was clear that any such preprocessor must be capable of acting as a C++ frontend itself and consequently as a source-to-source translator, due to the interactions that are possible between C++ and a plugin. The authors are unaware of any sufficiently sophisticated translator in existence which would not restrict us in the future. For example, clang [18] does not yet support templates, and the ROSE [10] framework uses the EDG C++ frontend, whose license does not permit redistribution of changes. We thus implemented the extension using a specific frontend until suitable tools become available.

An additional design decision was that of the implementation language for plugins. Our initial choice of language was motivated by the functional and in particular the deterministic nature of template metaprogramming, which preserves the one-definition rule between multiple translation units. Haskell enabled us to retain template metaprogramming's pattern matching features, and expand on this via advanced techniques such as "scrap your boilerplate". The pure nature of Haskell has also enabled us to preserve safety via restricted monads, thus preserving the one-definition rule, and assuring the correct operation of GCC's garbage collector.

The implementation of a custom programming language for plugins (the strategy used by MELT [11]) was considered as this may improve integration between the compiler and the plugin language, and efficiency. However, this would severely increase development time with disproportionate benefits.

Performance We evaluated the compile-time performance of our framework using a side-by-side comparison of the Haskell implementation of `dual` against a C++ implementation on a 2.13GHz Intel Core 2 running Ubuntu 7.04. We measured the total compiler execution time for each implementation performing a fixed set of 1000 randomly generated dual computations using the "time" command. To measure the performance of the C++ implementation, we used a pristine version of the compiler. The fastest execution time over 6 runs for the C++ implementation was 4.605 seconds, and for the Haskell implementation 11.934 seconds. The results show that our extension's performance is acceptable (approximately 2.6 times slower than pure C++).

7 Conclusion and Future Work

We have presented a mechanism for compile-time computation using the Haskell programming language. Our work has extended the C++ type system using the Haskell programming language with a workable type representation and usage methodology. We also presented work on a program analysis framework for GCC. We have also presented a number of examples in the context of session types, and shown that our framework allows for a relatively simple implementation.

Our planned future work includes an implementation of typestates [19] for C++, and possibly all other languages supported by GCC, and an implementation of a bug checker tool for C++ and possibly all other languages supported by GCC, similar to Java's FindBugs [20] and PMD [21].

References

1. Collingbourne, P., Kelly, P.H.J.: Inference of session types from control flow. In: Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2008). (2008)
2. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices **38**(3) (March 2003) 26–37 Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
3. Steele, G.: Common Lisp: The Language. Digital Press, Newton, MA, USA (1990)
4. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. SIGPLAN Not. **37**(12) (2002) 60–75
5. Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K.: Design and implementation of metalevel architecture in C++ – MPC++ approach. In: Reflection '96. (1996) 153–166
6. Chiba, S.: OpenC++ 2.5 reference manual (1997)
7. Falconer, H., Kelly, P.H.J., Ingram, D.M., Mellor, M.R., Field, T., Beckmann, O.: A declarative framework for analysis and optimization. In Krishnamurthi, S., Odersky, M., eds.: CC. Volume 4420 of Lecture Notes in Computer Science., Springer (2007) 218–232
8. Quinlan, D.J., Vuduc, R.W., Misherghi, G.: Techniques for specifying bug patterns. In: PADTAD '07, New York, NY, USA, ACM (2007) 27–35
9. Khedker, U.: gdfa: A generic data flow analyzer for GCC. Retrieved from `http://www.cse.iitb.ac.in/grc/software/gdfa-v1.pdf` (2008)
10. Schordan, M., Quinlan, D.J.: A source-to-source architecture for user-defined optimizations. In Böszörményi, L., Schojer, P., eds.: JMLC. Volume 2789 of Lecture Notes in Computer Science., Springer (2003) 214–223
11. Starynkevitch, B.: Multi-stage construction of a global static analyzer. In: Proceedings of the 2007 GCC Developers' Summit, Ottawa, Canada (2007) 143–151
12. Collingbourne, P., Kelly, P.H.J.: A compile-time infrastructure for GCC using Haskell (extended version). Available from `http://www.doc.ic.ac.uk/~pcc03/grow09haskell-ext.pdf`
13. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. **11**(5) (2001) 467–492
14. Honda, K.: Types for dyadic interaction. In: CONCUR'93. Volume 715 of LNCS., Springer-Verlag (1993) 509–523
15. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
16. Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. **42**(2) (2005) 191–225
17. Wadler, P.: Linear types can change the world! In Broy, M., Jones, C., eds.: IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, North Holland (1990) 347–359
18. Lattner, C., et al.: clang: a C language family frontend for LLVM. Retrieved from `http://clang.llvm.org/`
19. Deline, R., Fähndrich, M.: Typestates for objects. In: ECOOP '04, Springer (2004) 465–490
20. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: OOPSLA '04, New York, NY, USA, ACM (2004) 132–136
21. Dixon-Peugh, D., Copeland, T., Le Vourch, X.: PMD. Retrieved from `http://pmd.sourceforge.net/`