Static Methods to Check Low-Level Code for a Graph Reduction Machine

by

Marco Polo Perez Cervantes

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

UNIVERSITY OF YORK

Department of Computer Science

November 2013

Abstract

This thesis is about checking code for a graph-reduction machine computing by template instantiation. An equation-based static checking method for low-level code is proposed in this thesis. The checking can be performed without requiring any extra code annotations. Most ill-behaved programs can be rejected and most well-behaved programs can be accepted. The template code has no explicit information about data types but the static checker works by inferring low-level recursive types. We show compatibility between high-level and low-level type systems. We evaluate empirically the effectiveness of checking to prevent failures. We investigate the low-level implementation of the static checker and how it can be made efficient.

Contents

List of Figures

List of Tables

For Isel, Iskay and Iyari.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Professor Colin Runciman for the continuous support of my PhD study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

I would like to thank the Plasma group for their insightful comments, and questions during the group seminars.

I would like to thank my Dad and my Mom, and all the members of my very big family.

Last but not the least, I would like to express my gratitude to Jenny, Ana, Alfonso, Luis, Michael and Matthew for their support and friendship during my stay at UK.

Declaration of Authorship

I, Marco Polo Perez Cervantes, declare that this thesis titled, 'Static Methods to Check Low-Level Code for a Graph Reduction Machine' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Chapter 1

Introduction

1.1 Introduction

In this thesis we are interested in the application of static checking techniques to low-level code such as Reduceron *template code* [1]. The Reduceron is a graph reduction machine that uses *template instantiation* model [2, 3], which can run on reconfigurable hardware called FPGAs to exploit the advantage of parallel computations.

Our work is not program verification, because we do not try to attest that a program produces the correct result. In addition, we are not using disassembling techniques to obtain and check a source-level representation.

The importance of low-level code verification is discussed in Clutterbuck's [4] work and also in Proof-Carrying Code [5], and Typed Assembly Language TAL [6]. More recently, and more similar to our approach, are the type-inference systems described in [7] and [8]. The scope of the techniques presented in this thesis is not limited to the static checking of template code for Reduceron. Similar methods might be applied to other forms of byte-code or compiled-code.

During the compilation process, F-lite code is transformed to equivalent template code (See §2 [1]). A well-established compiler might be trusted to produce safe code. However, template code could be *modified* at some stage after compilation and before execution, or template code might be supplied from some other source. Faulty code could produce a failure at run-time such as an invalid memory access.

Our idea to is protect the Reduceron code against such failures by *examination of the low-level code of the Reduceron*. Taking the ideas of PCC [5] and TAL [6, 9] in which mobile code is verified before its execution, we investigate here how to build a type-checking system to guard against unsafe template code such as code modified inappropriately by hand, or code intercepted and altered before it reaches the target machine. Unlike [7] and [8], we do not use any extra annotations but infer types directly from template code.

1.2 Thesis Statement

Here we present the claims for our thesis.

1. Type-Checking methods can be used as the basis of a static checker for low-level code such as Reduceron *template code*. The checking can be performed without requiring any extra code annotations.

2. In experimental tests using mutated programs most ill-behaved programs can be rejected and most well-behaved programs can be accepted.

3. The low-level types inferred during static checking of typeless template code can be shown compatible with translated high-level types in source programs.

4. Checking can be made efficient in the following sense : the time and space complexity can be reduced by translating the model into an implementation, giving the first step towards further optimisations.

1.3 Thesis Rationale

- *Type-Checking methods can be used as the basis of a static checker for low-level code such as Reduceron template code.* In the high-level source, several static methods can be used to prove with rigour that source program satisfies some requirements. However, the problem arises when the compiled code is sent to the code consumer by using an unreliable channel. Who can tell us that the code will respect the invariants of the high-level specification? We can choose a complex method to check the template code, for

instance requiring a certificate from the code producer guaranteeing that the code comes from a *trusted* producer. This approach depends on the code producer and moreover, it only attests that the program is provided from a trusted source, but not that the code is well-behaved. By type-checking the template code we can catch type errors before the template code is executed. This method provides a feasible way to check if some template code is well-behaved. However the type inference for low-level code is a challenging task, at low-level for instance there is much less information for algebraic data types than its high-level representation. Although we have decided to start with a type-checker there are other static checking methods based on data flow analysis [10, 11] which can be combined with type-checking. We refer the reader to Chapter 2 in [12] for further details in verification techniques for low-level code.

The checking can be performed without requiring any extra code annotations. Although no explicit type information is provided in the low-level code, it is feasible and possible to infer type information. Here, it is important to note that our work is focused on *low-level code without any extra information.* The opposite approach to our work is in [13–16], where the source-level code is checked or verified by using extra information.

- *Most ill-behaved programs can be rejected and most well-behaved programs can be accepted.* Taking as basis the statement of Milner : "Well-typed programs cannot go wrong" [17] , we can measure the effectiveness of the type checking by comparing the static checking and the operational semantics. Ideally, the well-behaved programs are well-typed and the ill-behaved are ill-typed. In Chapter 4 we measure the extent to which this ideal is achieved.

- *The low-level types inferred during static checking of typeless template code can be shown compatible with translated high-level types in source programs.* The type compatibility between the high-level and the low-level type systems is expressed in the properties of the solutions for the system of equations in both systems.

- *Checking can be made efficient in the following sense : the time and space complexity can be reduced by translating the model into an implementation, giving the first step towards further optimisations.* As a starting point we

could take advantage of the single-threaded data structures in the implementation to have a gain in the efficiency and in the reduction of the costs of memory.

1.4 Contributions

The main contribution is the use of type-checking methods to check statically low-level code, and prevent the occurrence of *run-time* failures. In addition, this static checking can be efficient, using a minimal amount of resources in the run-time system. Our contributions are:

- To avoid the need for dependent types, as used in some other static checkers for low-level code [7, 8], we apply a primitive analysis before the type inference phase. We call this analysis *PrimCheck*, and it eliminates the primitive applications from template code. The details are in Chapter 4 §4.3.
- Our second contribution is the way of viewing the type inference problem as a problem of solving a system of equations [18]. We can extract the type information from *untyped compiled code* such as Reduceron template code. In §4.4.4 of Chapter 4 we explain this idea. We combine the ideas of Cardelli's and Colmerauer's work in [18, 19].
- We evaluate *empirically* the effectiveness of checking to prevent failures.
- We show compatibility between high-level and low-level type systems.
- We investigate the low-level implementation of the static checker and how it can be made efficient.

1.5 Roadmap

In Chapter 2 we give a general background of some of the static checking techniques for low-level code. We start the discussion with type checking and type inference approaches [18, 20], we continue with Java byte-code verification [21–25], and we end with Proof-Carrying Code [5] and related techniques [6, 26–32].

In Chapter 3 we give a description of a framework for testing, constructing a tool set to measure *empirically* the effectiveness of the static methods proposed in this thesis. We start with the description of the Reduceron machine, then we follow with the approach of *mutating* [33] the programs to be measured. Finally, we explain how we classify the results of our experiments.

In Chapter 4 we describe a type inference system for Reduceron code. Without using any extra information, we derive a system of type equations from Reduceron template code. We call this model Tyre-Check. We split the static checking into three phases which are described in the following order:

- *AtomCheck*, is a sanity check for template code.
- *PrimCheck*, a primitive analysis to eliminate the application of primitives.
- *TyreCheck*, the actual type inference.

In addition, by using mutated programs we evaluate *empirically* the effectiveness of checking to prevent failures.

In Chapter 5 we investigate the type-compatibility between a high-level type system for F-lite source program and the low-level type system for Reduceron code. Here, we describe the main parts of the type system for F-lite, then we explain the main points of the translation from high-level types to low-level type-terms equations. Finally, we show empirically that inferred low-level types are compatible with translated high-level types.

In Chapter 6 we explore issues of efficiency. In particular, we give an implementation in C language for the Haskell model. We give the description for the Haskell model –TyreCheckH, then we measure the time and complexity of this model. Then we explain the main points of the translation from our Haskell model to the C implementation – TyreCheckC_0. We measure the TyreCheckC_0 in terms of time and space to detect the components of the checker requiring the greatest memory allocation and the most time-consuming functions. Finally, we give an improved version of our C implementation which we call TyreCheckC_1.

In Chapter 7 we state our overall conclusions, and identify the possible lines of future work.

Chapter 2

Literature Review

The purpose of this Chapter is to give a general idea of the static checking and verification techniques. As our work is based on static checking and not verification (eg. program correctness), we explore the related techniques to static checking in more detail.

In §2.1 we give a brief background of the verification and checking of low-level code. In §2.2 we explore the literature related to types. In §2.3 the verification and checking of byte code is explored. In §2.4 we give general descriptions of Proof-carrying Code, Typed Assembly Language and related techniques. Finally, in §2.5 we give a summary of the techniques presented in this chapter.

2.1 Introduction

The program verification problem is not a new topic in Computer Science. This problem has been studied by some well-known computer scientists (Floyd, Hoare et al) [34–37] in the 60's and 70's, giving the foundations for research in this area for high-level code. Even if those techniques were originally conceived for high-level code, they could be applied to verify and check low-level code.

Late in the 80s Clutterbuck and Carré pointed out the importance of the verification of low-level code [4]. Nowadays, it seems to be important to verify low-level code due to the ubiquity of software. The massive networks in which code is in the form of binaries are a jungle where the code can be trapped and modified maliciously. Another application of the low-level verification is when we need

to perform some hand-coded optimisations in installed software: in the scenario where the raw code does not contain information or a mechanism to prevent the introduction of erroneous code. We cannot know if the code is safe according to the high-level specifications. More recently the problem of low-evel code verification has been studied by Myreen and Gordon [38, 39].

In this review we explore some related techniques for static verification of mobile code. For our purposes, mobile code is defined as a piece of code produced by a different computer or compiler (e.g., untrusted code producer) from the computer where the code is going to be executed (e.g., host machine).

In general, the mobile code is transported as assembly code or bytecode from one agent to another over an unreliable network. It is executed by the run-time system in the host computer. Some of the techniques described here, have in common that the hard part of the verification process is done by the *code producer*, at compilation time. Then the *code consumer* at loading time checks the *untrusted code* by using a light-weight verification algorithm. This checking guarantees the code is well-behaved according to the safety policies of the host run-time system. The spectrum of low level-code is broad, it can include bytecode, assembly code or machine code. For our work, we use these three models of low-level code interchangeably.

2.2 Type Systems

One discipline that addresses program static verification in a formal way is *type systems* [18, 20, 40]. Type systems can be used at high-level as well as in low-level code. This discipline guarantees at compile time that programs will not fail because of typing conflicts during the execution of the program. Informally, one view of a *type system* is a system to prevent the occurrence of execution errors in a program at run-time [18].

One of the first computer languages that had a *type system* was FORTRAN [20], establishing distinctions between floating-point and integer numbers. The study of *type systems* has passed from an informal discipline to a formal discipline.

According to Cardelli [18], type systems should be:

- Checkable. By using a *decidable* algorithm (called *type-checking*), the programmer can verify that a given program is well-behaved. If checking fails, the programmer catches possible errors before they happen.
- Transparent. The type system will report an evident reason for the fault, in case of a type-check error in the program.
- Enforceable. The type declarations, and the associated programs or functions, should be routinely verified.

2.2.1 Type-Checking and Type-Inference

There are two techniques that implement a type system. *Type-checking* attests that the type *information explicitly given* by the programmer is consistent with the language definition constraints. On the other hand, *type-inference* algorithms calculate the type information without any extra type signatures, they use an inference system to determine the type compatibility between expressions. We could have the combination of both algorithms, if we provide some hints to make the inference or the checking fast. In our work we use the type-inference technique.

The *type-checking* and *type-inference* algorithms are core components for the implementation of a type system. One type system can be implemented by several type-checking or type-inference algorithms. Such algorithms verify or infer that the operations in functions correspond to their type abstractions. They can check, for example, that a *sum* operation is performed only over numeric values; or they can guarantee that a concatenation of a list and a pair is not performed. Type-checking is part of the compilation stage, and in its origin, it was conceived to work with high-level representations of computer languages. Possible errors related to a mismatch in types can be detected before the program is running. According to Milner, an important design goal for the systems is that if the program code respects the typing rules imposed in the type system, then the program "will not go wrong" [17] during execution time.

If some type information is provided then the type-checker will detect if there are inconsistencies between the provided signature and the inferred types. We may view type checkers as theorem provers but offering more information if an error occurs during the rule derivation.

2.2.2 Polymorphism

The main idea of parametric polymorphism [41] is that we can define generic functions to avoid the expansion of the size in a given program. For instance, consider a function to compute the addition of two numbers. We can define several functions to deal with integers and real numbers. By using polymorphism, we can write one function with generic parameters that allow the inclusion of integer and real parameters.

One of the basic principles in type systems with polymorphic types is the idea of *unification*. This algorithm decides if a pair of type expressions are compatible. By unifying pairs of type expressions we can get the *most general unifier* [42]. More precisely, a finite mapping from type variables to types. The connection between Robinson's unification algorithm [42] and polymorphic type systems was discovered by Milner [17].

The idea behind type-inference is to reconstruct the types of the expressions. It can be viewed as a two stage process. In the first stage, the primitive types (of the abstract syntax tree) of an expression are introduced. In this part of the process the unknown types for the expressions are labelled as new variables. In the second stage, the types for the unknown types are reconstructed or calculated using type-inference rules. This last part involves the unification of the equation types in order to get the most general type. The typing rules are applied recursively to the abstract syntax tree. At the end of the process (if it terminates), we obtain from the *context* the types for every single expression in the program.

Most modern functional languages make extensive use of types [43–45]. By using strong type systems in functional programming we can generate type-safe components of software. For example by using type signatures in functions the programmer can catch type errors at compile time, thus improving the productivity of the software development process. In addition, type signature provides a good way of documenting the code [20]. The lack of side-effects in the declarative style provides a strong mechanism to construct programs and reason about their properties. Moreover the size of the programs can be drastically reduced by using higher-order functions and lazy evaluation [46]. Pierce and Cardelli's work provides a full and deep coverage of type systems and their relation to programming languages [18, 20].

2.2.3 An Example of Type Inference

In Figure 2.1 we have the *length* function defined as two pattern matching definitions in 2.1 and 2.2. In the remaining part of the Figure 2.1 we depict briefly the type equations to infer the type for *length* function. Before we proceed with the type equations explanation, we recall that the type of *length* is $[a] \rightarrow Int$. That is, a function that takes a polymorphic list of type $[a]$, and returns the length, which is of type Int.

Now we describe the type equations informally, and how we can discover the type information. To start, we introduce an equation to denote the type of *length* in Equation 2.3. At this moment we only know that a function *length* is a function with type $a \rightarrow b$, from type a to another type b. In the same way we establish the type equations in 2.4 and 2.5 for the two pattern matching definitions of *length*. From the argument of definition 2.4, we discover that the type variable a_0 is a list of some type a_2, we denote this in Equation 2.6. From the right hand side of 2.4 we can see that the type of b_0 is Int, and we denote it on Equation 2.7. For the second definition 2.5, we extract the type of the argument, we discover that the type variable a_1 is a list of some type a_3, we denote this in Equation 2.8. The most interesting part is in the right hand side of 2.5. There are two function applications, the first one is the function $(+)$ applied to two arguments of type Int in Equation 2.9, the first argument is the numeric constant which is of type Int. The second argument is the application of the function *length* itself to the remaining list xs, we introduce a new type variable b_2, and we add the constraint that it must be of type Int in Equation 2.10. Apart from that, the variable b_2 must be equal to the application of *length* to the remaining list denoted in type $[a_3]$, that is denoted in Equation 2.11. In Equation 2.11, notice the type of the *length* function which is a *copy* and not an instance of *length* type. We need a copy of the type because the type must be the same when we have a recursive call. In Equation 2.11 we introduce a new type variable b_3, which will be the type result of that function application. The last equation is 2.13 and that constraint $l_0{=}l_1$ guarantees that the two definitions are type compatible. Once we solve these equations we obtain a system of equations, where the type of the *length* function is denoted by Equations 2.14, 2.15, 2.16, and 2.17.

Definition of *length* :

$$length[\,] = 0 \tag{2.1}$$
$$length(x : xs) = (+)\ 1\ (length\ xs) \tag{2.2}$$

Type equations :

$$l = a \to b \tag{2.3}$$
$$l_0 = a_0 \to b_0 \tag{2.4}$$
$$l_1 = a_1 \to b_1 \tag{2.5}$$

$$a_0 = [a_2] \tag{2.6}$$
$$b_0 = Int \tag{2.7}$$
$$a_1 = [a_3] \tag{2.8}$$
$$b_1 = Int \tag{2.9}$$
$$b_2 = Int \tag{2.10}$$
$$a \to b = [a_3] \to b_3 \tag{2.11}$$
$$b_3 = b_2 \tag{2.12}$$
$$l_0 = l_1 \tag{2.13}$$

Solving type equations:

$$l = a \to b \tag{2.14}$$
$$a = [a3] \tag{2.15}$$
$$b = b_3 \tag{2.16}$$
$$b_3 = Int \tag{2.17}$$

FIGURE 2.1: Example of *length* function and its inferred type equations.

2.2.4 Recursive Types and Subtyping

Recursive definitions are present in most programming languages. For example in functional programming languages such as ML or Haskell data structures like trees or lists are instances of recursive types. In section 2.2.3 we saw that it is possible to handle recursive type information by means of equations containing recursive references in their subterms: we reuse the idea of Abadi's and Cardelli's work stated in the following slogan: *"Recursive types can hence be described by equations, and we shall see that in fact they can be unambiguously defined by equations."* [47].

The problem of subtyping is a whole area, and it has been studied by Cardelli, Mitchell et al [47–55].

2.3 Static Checking of bytecode

Much of the work on low-level verification of bytecode has been prompted by the development of Java bytecode which is executed by the Java Virtual Machine JVM [56, 57]. Probably this tendency is due to the popularity of Java and the use of architecture-independent design. The same Java code should run everywhere without changes. In addition to that, any language could potentially be compiled to Java bytecode, allowing more flexibility for the programmer.

Java bytecode was designed to be portable and statically checked [58]. The proliferation of many architectures forced the programming language community to think how to write different code for each specific platform. The price of this portability is the vulnerability of the host machine, as Java bytecode can be downloaded from an untrusted source. Hence the machine needs some mechanisms to prevent ill-behaved programs. Gosling was inspired by the idea of bytecode from UCSD Pascal which used a primitive bytecode called p-code and ran in a virtual machine. *"The solution we chose was to compile to a byte coded machine independent instruction set that bears a certain resemblance to things like the UCSD Pascal P-Code"* [58].

Java bytecode is a sequence of *opcodes*. Some opcodes contain implicit type information in the instructions. For example, in Figure 2.2 a small subset of instructions is illustrated and denoted by `instr`. The opcode `iload n` is the instruction that loads an int value `n` from a local variable, the opcode `aload n` loads a reference from a register `n` onto the stack, the opcode `astore n` stores a reference into a register, opcode `iadd` adds two ints from the top of the stack, the instruction `goto l` jumps to another instruction in the set of instructions denoted by `l`, `iconst n` loads the numeric constant `n` into the stack, and finally, the `ifeq l` opcode is a conditional jump, if the local variable is zero then it jumps to the instruction at label `l`.

Static analysis techniques for Java bytecode verification [25, 56] are well known in the bytecode verification community. Model checking, abstract interpretation, type-checking , or the combination of some of them, are the main methods used

```
instr :: = iload n
         |  aload n
         |  astore n
         |  goto l
         |  iadd
         |  iconst n
         |  ifeq l
```

FIGURE 2.2: A small subset of Java bytecode instructions.

to deal with the problems of security in the JVM at load time. The first approach to stop ill-typed programs was due to Cohen [23], who proposed dynamic checking in a so-called *defensive Java Virtual Machine (dJVM)*. Cohen used ACL2 [59] as a language to specify and validate the model. However, in practice the cost of checking at run-time was too high.

2.3.1 A Machine over Types

To alleviate the problems of dynamic-checking Java bytecode verification, some ideas of *static checking* came in the 90s. The first implementation was an abstract machine that uses *data flow analysis* (DFA) combined with type-checking, developed by Yellin and Gosling at Sun [58]. Basically, this abstract machine checks the type information of every instruction in the code. It works over types instead of concrete values. In addition to type-checking it makes pertinent checks to detect possible underflow or overflow in the stack of type instructions and in the stack of type registers.

The static checking of Java bytecode includes several stages in the work of Yellin and Gosling, in this literature review we are only interested in two points: *The code is free of stack overflow and stack underflow*, and *It is type safe*. The Java bytecode is safe if the following constraints are true:

- The code is free of stack overflow and stack underflow.
- It is type safe.
- The program counter is in the scope of the method.
- The code has a valid register initialisation.
- It has the objects initialised before they are used.

$$[\![iconst\, n]\!] < S, R > \Rightarrow\ < int : S, R >$$
where
$S < MaxStackSize$

$$[\![astore\, n]\!] < t : S, R > \Rightarrow\ < S, R[n \backslash t] >$$
where
$0 <= n < MaxRegSize \wedge t <: Obj$

$$[\![iadd]\!] < int : int : S, R > \Rightarrow\ < int : S, R >$$

$$[\![iload\, n]\!] < S, R > \Rightarrow\ < int : S, R >$$
where
$0 <= n < MaxRegSize \wedge Rn = int \wedge |S| < MaxStackSize$

$$[\![istore\, n]\!] < int : S, R > \Rightarrow\ < S, R[n \backslash int] >$$
where
$0 <= n < MaxRegSize$

$$[\![aload\, n]\!] < S, R > \Rightarrow\ < Rn : S, R >$$
where
$0 <= n < MaxRegSize \wedge Rn <: Obj \wedge |S| < MaxStackSize$

FIGURE 2.3: Example of abstract interpretation rules in Yellin and Gosling's static checker for Java bytecode.

The JVM is a stack machine in which the instructions are popped or pushed onto the stack during execution. One way to verify that the code is safe, is to use a model to *approximate* execution on a defensive *dJVM*. Here is where the abstract interpretation machine enters into the scene. Instead of working over concrete values it operates over types. Basically this method is a combination of data flow analysis *DFA* and type-checking.

The abstract interpreter checks that every instruction before its execution satisfies some specified properties in the form of constraints. These constraints can be related to types or memory bounds. The safety of the byte code can be guaranteed at linking time or loading time, so checking is done only once. Once the verification has been performed then the normal JVM can execute the bytecode as many times as needed.

More formally, the abstract interpreter is a transition relation that operates over types of instructions and registers [25], and it is denoted in Figure 2.3 as

$$[\![instr]\!] < S, R > \Rightarrow\ < S', R' >.$$

Here *instr* is the instruction, S is the stack of types for instructions, and R is the stack of types for each registers. S' and R' are the new stacks, for instruction and register types. Figure 2.3 illustrates an example rules for common instructions used by the this transition relation model.

The rule *iconst n* simply pushes an int type on top of the stack denoted by *int:S*; in addition it checks that the current size of the stack is less than the maximum stack size *MaxStackSize*. The second rule, *iadd*, takes two *ints* from the top of the stack and pushes an *int* type on top of the stack denoted by *int:S*. When the instruction *iload* is evaluated in the abstract machine, it checks that the register at index n is of type *int*, also checks the bounds of the register n and the size of the stack S. If all the checking is correct then it pushes a type int on top of the stack. The instruction *istore* stores int type from the top of the stack *int:S* into the type register R n. Before this substitution is performed it checks the limit of n to be consistent to the size of R, $n < MaxRegSize$. The instructions *aload* and *astore* have similar behaviour, but instead of checking that a type is an *int* type, it looks for a more generic type of type Object, in the stack S and in the record type set R. In all, there are more than 200 instructions for the JVM, here we have presented the rules only for six of them to illustrate how the abstract machine for types works.

If there is no transition for some state then the machine is stuck. For instance, the *istore* rule assumes that the top of the stack must be an *int*. If we have an opcode *istore* and the top of the stack type is not compatible with *int* then there is no meaning for that state. It is important to say that the verification consists of several steps, the abstract interpreter is applied only after previous checking for sanity, and method initialisation. Here we only present some rules for the verification of instructions inside a method; there are transition machines for methods, and classes. For calculating types of objects it uses a subtyping relation $<:$, where classes are subtypes of *Object*.

This model of verifying Java bytecode by combining DFA and type-checking is the most common. However, the specification for JVM lacks the required formalism, and some inconsistencies are found in the specification [22].

In order to provide a specification for JVM as the basis for a static method to check the bytecode, a new formulation was proposed by Klein, Nipkow, Quian, Coglio et al [22, 24, 60, 61], based on theorem provers such as HOL and Isabelle.

Moreover, this formalism can be used by others to explore the properties of the verification techniques or the JVM. The work was proven to be valid for a non-trivial subset of byte code instructions, with mechanically checked proofs, offering some guarantee of reliability.

2.3.2 Subroutines

The verification of Java bytecode is a hard task. One of the biggest problems is verification of subroutines. They are problematic because subroutines share the same stack frame as the method calling them; because of this *sharing*, the addresses when a subroutine is called can have a polymorphic type. The main reason of having subroutines in Java is to support the isolation of the code. The classic example is the $try - catch - finally$ block to handle exceptions when a problem arises during the execution.

The subroutine verification problem for bytecode was first addressed in the work of Stata and Abadi [21] at the end of the 90s. Klein developed a verified extended checker for subroutines by using models in HOL/Isabelle in [62]. A complete section in Leroy's survey [25] reviews the history of subroutine verification problems, and some solutions to handle this problem.

2.3.3 Lightweight Bytecode Verification

Another branch of bytecode verification is Lightweight Bytecode Verification (LBV), whose main purpose is to verify Java Smart Cards [63]. The proliferation of small devices which execute untrusted byte code, brings new challenges into the scenario of bytecode verification.

One of these problems is the small amount of resources available in such devices. Under these conditions standard bytecode verification algorithms are not feasible, due to the large amount of computation and memory usage. The first attempt to verify code in small devices was proposed by Rose et al [64, 65]. The main idea of LBV is to decouple the problem of the verification in two stages:

- At *compilation* time a certificate (the output of the data flow analysis (DFA)) is *created*,

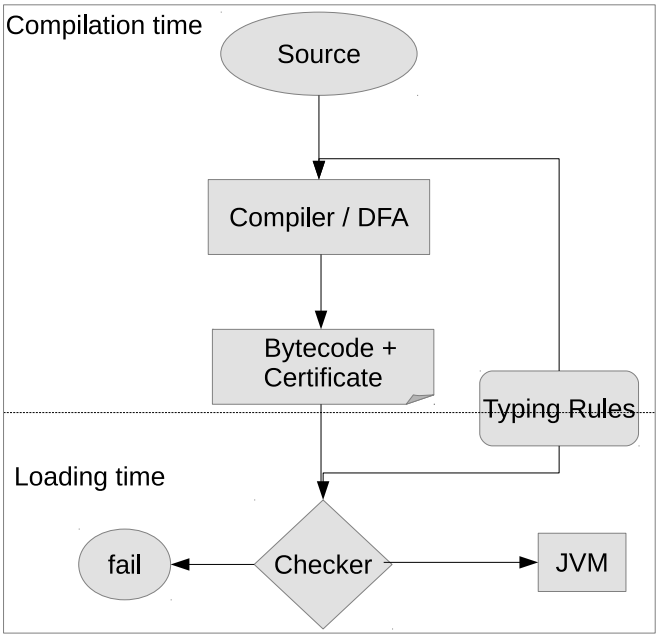


FIGURE 2.4: Lightweight bytecode verification diagram.

- At *loading* time the certificate (only the needed information) along with the code is *checked* in the small device.

This approach reduces the computation steps required in the smart cards to validate the bytecode. All the hard work has been done off-line, with results in the form of a certificate.

The diagram in Figure 2.4 describes the overall mechanism of . In the *compiler side* we have the *DFA* which produces a *Certificate*. This certificate makes use of the typing rules for Java byte code. The *Checker* on the smart card side, takes the certificate and validates it by reconstructing only the information that can be obtained in one single pass in the bytecode. If the proof is correct then the bytecode is sent to the *JVM* to be executed. Notice that the *DFA* and the *Checker* make use of the *type information* reconstructed and assigned to each instruction and stack frame by the LBV.

2.4 PCC and TAL

2.4.1 Proof Carrying Code

Proof Carrying Code (PCC) [5] was proposed by Necula and Lee in 1997. The aim of PCC is to provide a *safety proof*, along with the *binary code*, to the host system. This proof is generated by the compiler or code producer, and attests that the attached code respects the security policy provided by the code consumer. Formal logic is used to express this *safety policy*. The PCC technique is based on the idea of having almost all the process of verification performed by the producer. All the code consumer has to do, is to check "off-line" the proof attached to the untrusted code. Once this checking process is performed, and if the check succeeds, the code consumer can execute the code without using extra verification tools. So execution time is not increased.

In the PCC system, everything is centred around the *safety policy* [5]. The safety policy (shown in Figure 2.5) is provided by the code consumer. Proof that a safety policy is respected involves three components : a set of rules(e.g., type-inference rules), the verification condition generator VCGen (producing Floyd-Style predicates [34]), and the interface which is basically a set of preconditions and post conditions describing the invariants before and after the function calls. VCGen takes as input the safety rules, the assembly code, and the preconditions and post conditions to compute a set of safety predicates in first-order logic. The verification of this safety predicate is the proof that the code respects the safety policy given by the code consumer. The verification is performed by a theorem prover. The proof is encoded in a subset of Logical Framework (LF) along with the assembly code, and sent forward to the code consumer. The code consumer uses a type-checking algorithm for LF representation, to verify that the proof is safe. If it type-checks, then the code respects the safety rules provided by the consumer of the untrusted code.

Here we list some of the main advantages of the PCC technique :

- The verification of the proof is faster than its production. A fast type-checking algorithm is used to prove the correctness of the proof attached to the binary code.

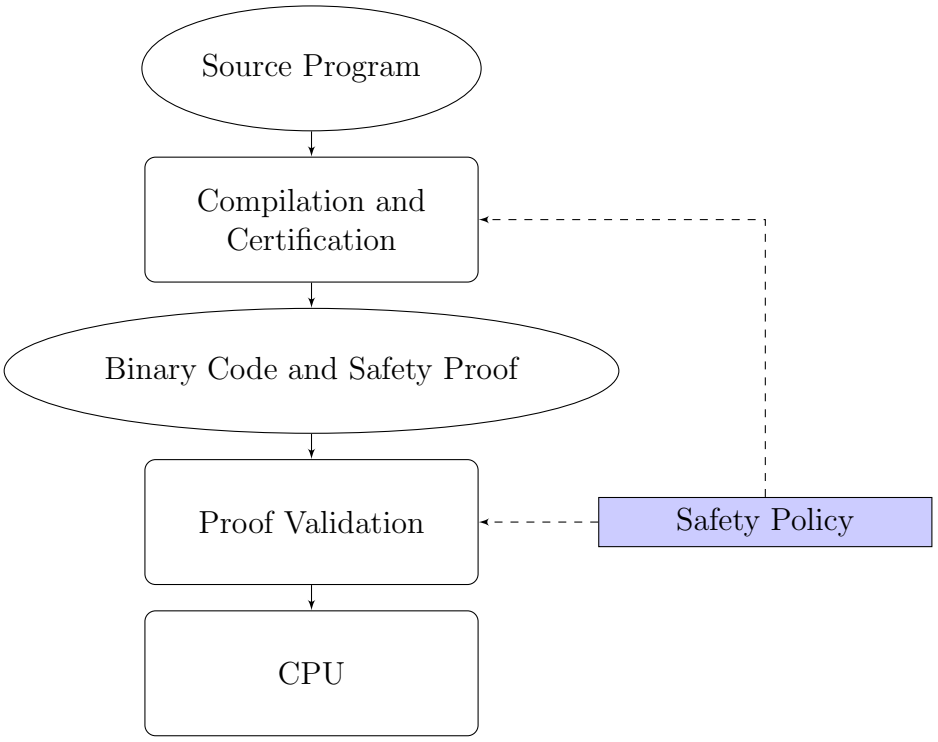


FIGURE 2.5: Proof-carrying code diagram.

- PCC is tamper-proof, any alteration in the binary code or in the proof is trapped by the verification process. As analogy with types, we can see the safety proof validation as type-checking. If we modify the code the program then it might be not longer a valid typed program, in the same way, if we modify the safety proof, then the validation results in an error. In the case that the safety proof and the code are modified, if the proof respects the safety policy then the program is safe, even if it is not correct.

- Common techniques can be used across different programming languages. First-order logic provides a way to prove the correctness of a given program independently of the language used to construct the program. The experiments in the original PCC were focused on verifying type information and memory safety.

- PCC verifies the correctness of code. It avoids reliance on trusted producers (e.g., a certified compiler). It is a general framework that can be used even if the source of code is untrusted.

- This technique alleviates the problem of the verification on a target machine with few resources, since the heavy part of verification is performed by the producer.

The main disadvantages of PCC are:

- The size of the proof can be very big in comparison to the size of code.
- The VCGen is a big piece of code, the verification of such piece code is a challenging task. A bug in the VCGen could end in a wrong verification condition predicate.

2.4.2 Typed Assembly Language

A related technique to PCC is Typed Assembly Language (TAL) [6], which was developed in 1998 by Morrissett et al. The basic idea in TAL system is that the compiler can insert type information in the form of labels in the assembly code. The approach of TAL systems is to preserve the typing information in the process of compilation, from the source code to the assembly language. If the assembly code type-checks then the code respects the operations or rules previously given by the code consumer in the form of typing rules. This technique is less robust than PCC, in the sense that it only tackles the problem of type safety.

Some compilers using this technique were developed, each of which preserves typing information from source code to intermediate stages, such as closure conversion or lambda lifting. Figure 2.6 depicts a staged transformation from System F (λ_F) to Typed Assembly Code (RISC instructions) as follows:

- The intermediate language λ_F is transformed to λ_K which uses Continuation Passing Style. In this part of the transformation, the intermediate language has continuations, instead of returning values the functions apply a continuation to them.
- The transformation from λ_K to λ_C and λ_H is in two steps. The first one is a simplified closure conversion; any variable from the context of the function must be transformed to additional arguments of the specific function. The second step is a lambda lifting process, in which all the local definitions of functions are hoisted by using functions and parameters.

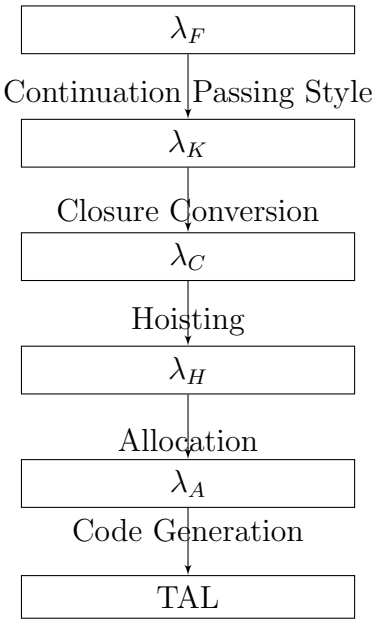


FIGURE 2.6: TAL transformation diagram.

- The transformation from λ_C and λ_H to λ_A is called allocation and produces the intermediate language with "lets" as constructors, these let instructions represent allocations in memory. Initialisation flags are added to every field of tuples defined in the previous stage.
- The last transformation from λ_A to TAL generates the TAL code. A simple type-checker algorithm checks the TAL code off-line.

Another implementation of TALx86 [26], is a realistic typed assembly language, sufficient to implement a subset of typed C called Popcorn. Many properties can be checked using typing information, those properties include memory address allocation, types of the variables, stack-allocation, and basic type constructors (e.g., arrays and tagged unions).

The TALx86 instructions are a significant set of INTEL IA32 (32-bit 80x86 flat model) assembly language, to be executed on Intel Pentium processors. Table 2.1 identifies the main components involved in the TALx86 system.

TALx86 uses MASM syntax for data and instructions. The data is extended to handle the type annotations inserted in the code. The type preconditions in form of annotations, are used to specify the types of the instructions before the control of the code is passed from one address to other. These kind of annotations are of

TALx86 tools	
talc	Type-checker for TALx86 code.
link-verifier	Linker for TALx86. Verifies that the linking of TALx86 files is safe.
assembler	Assembles a TALx86 code to produce the object file (COFF or ELF format)
popcorn	Subset of C that compiles to TALx86.

TABLE 2.1: Main components of TALx86 implementation.

```
-- PopCorn code

 int i= n+1;
 int s= 0;
 while (--i> 0)
          s +=i;

-- TALx86 code

    mov   eax,ecx          ; i=n
    inc   eax              ; ++i
    mov   ebx,0            ; s=0
    jmp   test
body :{eax:B4, ebx:B4}
    add   ebx,eax          ; s+=i
test :{eax:B4, ebx:B4}
    dec eax                ; i--
    cmp eax,0              ; i>0
    jg  body
```

FIGURE 2.7: An example of Popcorn and its TALx86 representation.

the form : $\forall\alpha_1 : \kappa_1 ... \alpha_m : \kappa_m . r_1 : \tau_1, ..., r :_n : \tau_n$ where $\alpha_1, ..., \alpha_m$: are the bound type variables, and allow registers to have a polymorphic type. The annotation $r_1 : \tau_1, ..., r :_n : \tau_n$ says that every record from r_1 through r_n has the type τ_1 to τ_n respectively. The κ_1 and κ_m allow the possibility of having different "kinds" of types in the TALx86 implementation. The type-checker *talc* verifies that instructions respect these type annotations for a given piece of assembly code.

2.4.2.1 An example: The sum of the first n natural numbers.

Figure 2.7 shows Popcorn code to compute the sum of the first n numbers. The fragment of the corresponding assembly code is in the same Figure 2.7. This assembly code includes annotations for types in the *body* and *test* labels. In the label *body*, the type annotations $eax : B4$ and $ebx : B4$ mean that eax and ebx have type B4 (abbreviation of Byte 4). A similar situation occurs in the label *test* where eax and ebx are required to be of type B4. This simple example shows how the types are represented in TALx86.

Array-bounds checking is the one of most complicated aspects of TALx86. This is because the size of the array is unknown until the execution of the code. TALx86 uses two type constructors: the first new type constructor is $S(s)$, which is called singleton type, where s is an integer expression. The purpose of this new type is to assign an integer value to the register, for example, if ecx is represented as $S(4)$ then the value in ecx must be 4.

The second type constructor is $array(s, \tau)$ where τ is the type of the array elements and s is an integer expression representing the size of the array.

One main difference between TAL and PCC is that TAL applies a type-checking algorithm to the assembly code avoiding the use of proofs in a separate logical framework (LF). There is no need for a separate theorem prover. In this way, the size of the binary code is considerably reduced if it is compared against the binary code produced by PCC systems. Provided that types are preserved along the whole process of compilation, this technique generates safe assembly code automatically. The main advantage of this technique is the reduction of the *Trusted Code Base*. The verification process on the consumer side, requires only a type-checker for the typed assembly code.

Advantages of TAL include :

- The size of the binary code is reduced. It is not necessary to have a proof in a separate logical framework. The typed assembly language serves as source for type-checking tools.
- A Simple and fast type-checking algorithm is applied to the annotated assembly code. If the code type-checks then it is safe to execute. In practise

the actual code is kept separated from the type information, allowing to execute the assembly code in the standard way.

- The process to create typed assembly code can be automatic, by the preservation of the types during the compilation process.

The main disadvantages of TAL are :

- Any program that violates a type system's invariants will not be typeable under that type system, even if the code is actually safe.
- It is less powerful and general than the PCC technique. PCC is more general because it can be applied to check other properties not related to the type system.

2.4.3 Foundational Proof Carrying Code

Foundational Proof Carrying Code (FPCC) was developed around 2001 by Appel et al [27]. The main idea of FPCC is to reduce the Trusted Code Base (TCB) proposed originally by Necula in his PCC system. The main motivation of FPCC is to make the TCB as small as possible, without committing to any specific type system [28]. In the FPCC system, the VCGen is eliminated in order to reduce the TCB. Appel argues that the VCGen tool has too many lines of code (˜23,000 lines of C). Proofs may be wrong due to the presence of a bug in VCGen, resulting in a flawed verification process. In Figure 2.8, we can see that the only two elements of the TCB are the Checker and the Axioms and Architecture Specification (the safety policy). The models for types are independent of the safety policy, making FPCC a more extensible system than its counterpart PCC.

The main advantages of the FPCC technique are:

- Reduction of the Trusted Code Base. The elimination of VCGen reduces the size of the TCB, also the possible bugs inside the VCGen are eliminated. Instead of using first-order predicate logic the FPCC system uses higher-order logic predicates, which is a more powerful logic.

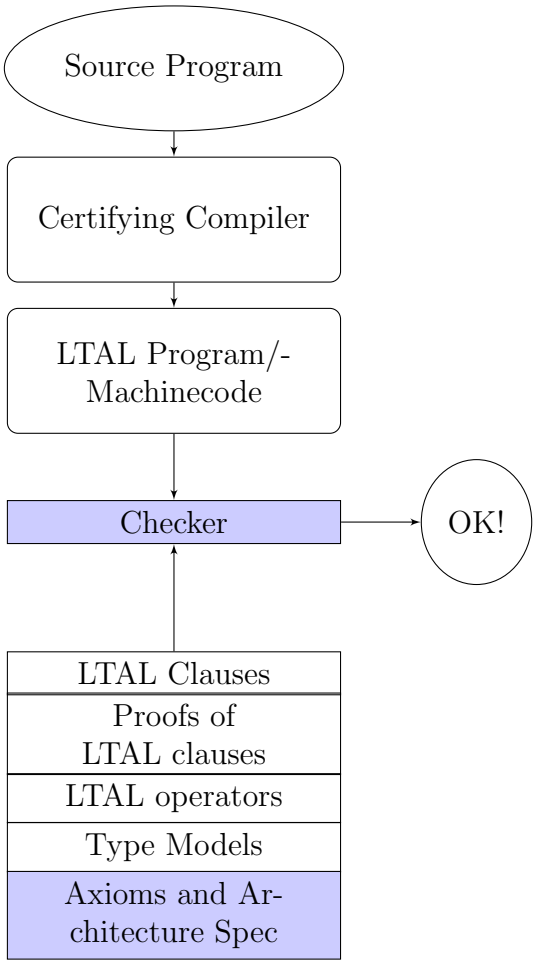


FIGURE 2.8: Foundational Proof-carrying code framework.

- FPCC uses the idea from TAL systems of producing assembly code with types in an automatic way, and the approach of the PCC technique of having a proof attached to the code that can be rigorously and mathematically checked. The proofs are generated and represented in Low-level TAL (LTAL), which is a representation of machine-code.
- The separation of typing rules from safety policy makes the FPCC system an extensible framework. With this advantage new models of types can be added without any alteration in the current system.

The main drawback of FPCC is:

- A big amount of the logic is moved from VCGen to safety proofs. Due to this situation the size of the proofs is greatly increased. The need for very large proofs is the biggest cost of having a more general PCC system.

2.4.4 Subsequent Developments Related to PCC and TAL

The three techniques we have reviewed were mainly developed in the years 1996-1999 (PCC), 1997-2001 (TAL), and 2000-2004 (FPCC). Several research lines have their foundations in those verification techniques. By using one of these techniques the run-time is not modified. This property makes these methods good candidates to verify programs in different environments without the need for a compiler or interpreter on the consumer side. This idea provides a reliable verification framework, even in small devices where the resources of hardware are limited. One of the biggest challenges is the size of the proofs, the proof must be informative enough and as small as possible in order to fit into the memory of the small devices where the low-level code is executed. In addition the trusted computing base must be as small as possible, due to the lack of power and data storage of the small devices.

Several research topics have been generated from these techniques [66–68]. Some researchers target the reduction of proof size (see [29]), others the exploration of new logic, and others the verification of other properties.

One application of PCC and FPCC is the project MOBIUS (Mobility Ubiquity and Security) [69, 70]. The project involved universities and software companies from around ten countries in Europe. The main goal of the MOBIUS project was to generate secure Java byte code for mobile devices in a fully automated way, by the combination of PCC techniques and JVM.

One particular characteristic of the MOBIUS project is that it uses types for information flow, so representing resources as well a classic types for code. The type information is checked by using lightweight algorithms in the consumer side. The problem of dealing with big certificates and reduction of the TCB are problems that appeared during this project. In essence MOBIUS makes uses of the two technologies related to PCC: type systems and program logic. This allows enforcement of other security properties such as user private data protection. The creation of the certificates follows a logical approach.

The claims for the MOBIUS architecture are:

- The digital proofs can be independently checked by users or third parties.
- Static checking of code avoids run-time costs.

- The techniques suit the real-world mix of mobile platforms.
- The solution supports developers who need to construct trusted applications.

2.5 Summary

The techniques for low-level static verification explored in the review of the literature are related. They verify low-level code and they use static analysis. Some additional information is attached to the low-level code, guaranteeing the safe execution of untrusted code without an overhead in execution time. In PCC a proof of safety is attached along with the code. This proof of safety is produced at compile time and is based in the safety rules provided by the code consumer. The code consumer verifies the proof by using a simple theorem prover, and if it holds then the foreign code is safe to execute. TAL system presents a similar approach. However, instead of using a separate proof of safety, TAL produces annotated code. This annotated assembly code is type-checked by a simple and fast algorithm in the run-time system. TAL can be seen as an instance of PCC in which only properties related to types are verified. Those techniques use a type-checking algorithms for a given program in a low-level code. In the case of PCC, it uses a theorem prover to verify the correctness of logical predicates. This mechanism presents a more general framework, because the properties are presented in first-order logic, allowing with this the use of any tool to attest the correctness of logical predicates. FPCC is an extension of PCC, the main goal is the reduction of the Trusted Code Base and the creation of a scalable framework to guarantee safety for low-level code. Additionally, FPCC introduces the LTAL concept, which is a low-level code for TAL. FPCC uses LTAL which is a lower representation of TAL, avoiding the problem of the verification of macro instructions (eg. malloc in TALx86). We refer the reader to a complete review of TAL related techniques in [9].

More recent work has focused on the reduction of the proofs or the reduction of the TCB, for instance the approach of Abstraction Carrying Code [30–32], provides another attempt to reduce the cost of the verification of the certificates, providing a framework that is based in logical high-order language. In this work, the authors propose to create small certificates based on fix point theory, and the checkers able to check in one single pass. Even though this technique was developed to deal with

high-level code it can be used for low-level code as in [32]. Some of the current work on verification of low-level code is based on the combination of different techniques, such as property based testing and type-checking [71]. The verification of Java bytecode techniques provides an avenue of ideas and problems to be solved when the bytecode is verified statically. In summary, the static verification techniques are *approximations* of the run time behaviours. So static analysis still leaves room for possible untrapped errors. In general, the gap between the approximations and run time execution can be reduced by providing information in form of assertions, type information, or data flow analysis outcomes.

Chapter 3

A Kit to Evaluate Tools that Check Reduceron Code

3.1 Introduction

In this Chapter we give details of the machinery to perform our experiments checking low-level code. Our experiments are based on Haskell models that abstract the behaviour of the actual machine for the Reduceron. In §3.2 we give an introduction to the Reduceron and we point out the connection to our work. In §3.3 we give a definition of the template code used by the Reduceron machine. In §3.4 we give a description of the Reduceron and its operational semantics. In §3.5 we give a definition of our model for execution of Reduceron code and the classification of the possible states where the machine fails. Then, in §3.6 we explain an automated approach to produce mutation variants of *template code*, and compare some alternatives to produce code. In addition we expose in a diagram the classification of the outputs for our experiments. With this classification we can measure the effectiveness of our static tools against the operational semantics. In §3.7 we describe the kind of mutations used in our testing. We give a classification for the test outcomes in §3.8. Finally in §3.9 we discuss the points of using the empirical approach to measure the effectiveness of our static checking.

3.2 Reduceron and Template Code

The Reduceron is an efficient graph-reduction machine based on template instantiation. When a functional program is evaluated, construction and deconstruction of expressions is performed in memory. On conventional computers, the use of the memory is limited by serial access. An alternative to solve this problem, is to evaluate functional programs on FPGAs where memory with parallel access can be configured as needed. Reduceron takes advantage of this technology, and in one clock-cycle can perform a complete reduction of a template application by accessing parallel memories for the template, the stack and the heap [72].

The Reduceron machine executes a kind of bytecode known as *template code*, which can be compiled from *F-lite*, a small functional lazy language described in [72].

Some aspects are particular to Reduceron template code: it uses *case tables* indexing alternative templates to encode case expressions, and it uses a non-standard applicative structure for primitives. For the purpose of our work the case tables and the primitives deserve further explanation.

Case tables. Case-expression alternatives in F-lite are compiled to functions by using the approach of Jansen/Scott encodings described in §2.4 of [72]. The idea of the case table is to replace many functional arguments by a single tabular argument to achieve more efficient evaluation.

Each case alternative in the case expression is represented in Reduceron code by a functional template $alt_i\ \vec{x_i}\ t\ \vec{v} = e_i$ where alt_i is applied to constructor arguments $\vec{x}$, a case table argument t, and free variables $\vec{v}$. The case table argument t is never used in the right hand side expression e_i.

Primitives. The Reduceron supports lazy evaluation, but the primitives need evaluated arguments. The problem is solved by encoding primitive applications in the form $arg_1\ (arg_0\ prim)$. With this arrangement the arguments are evaluated first. When the primitive is on the top of the stack the arguments are guaranteed to be evaluated.

Graph expressions and Templates. Each template can be viewed as a low-level encoding of a *let* expression. The let-bindings are encoded as *off-spine applications* and the body of the let as the *spine-application*. Each template $(arity, app, apps)$ in Reduceron template code is a tuple of $arity$, spine application app and off-spine applications $apps$. Instead of names the results of the off-spine applications are referred to by position (eg. PTR 0 for the first off-spine application).

How Reduceron is Related to our Work? Is it possible to check the low-level code statically before it is executed in the Reduceron processor?

In our work we will explore how to address the problem of static checking the *untyped* template code where the primitives and case tables are particular to this code. We shall perform the three static analysis in three stages: one for atoms, another for applications, and finally one for templates and how they are related.

The checking of Reduceron template code is challenging mainly because in the code there is no explicit type information: there are no declarations corresponding to high-level recursive data types. In addition, it uses particular encodings of case expressions and primitives which make it more complex to check than a standard applicative approach. Constructors are indexed by arity and position, but at low-level they are indistinguishable.

3.3 Template Code Syntax and Definition

The syntax of template code is defined in Figure 3.1. Template code for Reduceron can be generated, for example, by compilation of *Flite* [73] source programs.

The Reduceron code is self described in Figure 3.1, most of the atoms are common, the only exception is the constructor $TAB\ i$ which is used to encode case tables, see §2.4 in [1] for more details. Case tables are a special encoding in Reduceron template code for case expressions. The case expressions are transformed using Scott/Jansen encoding [74]. The case table constructor $TAB\ i$ contains the index i to the case alternative. The case subject evaluation index and the index i determine which alternative will be evaluated.

$Program$	::=	$\overrightarrow{Template}$	
$Template$	::=	$Arity \times Application \times \overrightarrow{Application}$	
$Application$	::=	$\overrightarrow{Atom}$	
$Atom$	::=	FUN a i	Function with arity a and address i
	|	ARG i	Reference to the i^{th} function argument
	|	PTR i	Pointer to the i^{th} application
	|	CON a i	Constructor of arity a and index i
	|	INT n	Integer literal n
	|	PRI s	Primitive function name s
	|	TAB i	Case table

FIGURE 3.1: Template code syntax.

For instance, in Figure 3.2 the code for append, which uses a case expression with two alternatives, is transformed to three function definitions, the first is for the top-level definition append, and the second and the third are the consCase and the nilCase alternatives. The $TAB\ i$ atom is used to encode <consCase, nilCase>, the index i will decide which alternative to choose depending on the evaluation of the case subject xs.

In the case alternatives the argument t is never used in the right hand side of the definitions.

```
--Flite Source
append xs ys = case (xs) of {
                Cons x xs -> Cons x (append xs ys);
                Nil       -> ys;
               }

--Scott/Jansen encoding
append xs ys       = xs <consCase, nilCase> ys
consCase x xs t ys = Cons x (append xs ys)
nilCase t ys       = ys
```

FIGURE 3.2: Jansen/Scott encoding for append function.

3.3.1 An Example of Reduceron Template Code

In Figure 3.3 we have the representation of the append source code and its translation to template code. As a convention, the first index of the templates in a given program is zero.

```
append xs ys = case (xs) of {
                Cons x xs -> Cons x (append xs ys);
                Nil       -> ys; }

main = append Nil Nil;

       (0,[FUN 2 1,CON 0 1, PTR 0],[[CON 0 1]])
       (2,[ARG 0, TAB 2, ARG 1],[])
       (2,[ARG 1],[])
       (4,[CON 2 0, ARG 0, PTR 0],[[FUN 2 1, ARG 1, ARG 3]])
```

FIGURE 3.3: Flite append function translated from Flite to Template code.

The template number zero *(0,[FUN 2 1, CON 0 1, PTR 0],[[CON 0 1]])*, is the main program, of arity zero. The atom *FUN 2 1* indicates a function of arity two and index one, *FUN 2 1* is applied to two arguments *CON 0 1* which is the encoding for *Nil*. The atom *PTR 0* is a reference to the off-spine application number zero, which contains the second argument of the function append.

Now, the template number one is the function append of arity two *(2,[ARG 0, TAB 2, ARG 1],[])*. In the spine application *ARG 0* is the case subject *xs*, the *TAB 2* atom is a reference to the case alternatives, for example if the case subject *ARG 0* evaluates to *Nil* then the second template will be evaluated. Otherwise, the third template will be evaluated. The atom *ARG 1* is the free variable used in all case alternatives.

The template number three is the recursive case alternative *(4,[CON 2 0, ARG 0, PTR 0], [[FUN 2 1, ARG 1, ARG 3]])*. The arity of this template is four, the reason is the special encoding which includes the two arguments *ARG 0* and *ARG 1* representing *x* and *xs*, an argument for case table and one free variable. The first atom *CON 2 0* is a constructor which expects two arguments and the index of this constructor is zero. The first argument is *ARG 0* and the second is a pointer the application *[FUN 2 1, ARG 1, ARG 3]*. This application corresponds to the recursive application at the high-level *(append xs ys)*.

3.4 The Reduceron Machine

The reduction machine has four memory areas: the program, the heap, the stack for reductions, and the stack for updates. These memories can be accessed in parallel, in the same clock-cycle. The execution behaviour of a Reduceron can be described by a small-step transition function from state S to state S', that is $S \rightarrow S'$. The state is a tuple $(Prog \times Stack \times Heap \times UStack)$ described in Figure 3.4, in each step a transition rule is applied to one state S to give a new state S'. The computation is stopped if the $Stack$ in the state S' is $empty$. The $Heap$ is modelled as a list of applications, and can be indexed by a heap-address. The stack is a list of atoms with the top stack element coming first and the bottom element coming last.

$$
\begin{array}{lcll}
Heap & ::= & \overrightarrow{App} & \\
HeapAddr & ::= & Int & \\
StackAddr & ::= & Int & \\
Update & ::= & StackAddr \times HeapAddr & \\
UStack & ::= & \overrightarrow{Update} & \\
Stack & ::= & \overrightarrow{Atom} & \\
State & ::= & Prog & \text{Program} \\
 & \times & Heap & \text{Heap} \\
 & \times & Stack & \text{Reduction stack} \\
 & \times & UStack & \text{Update stack}
\end{array}
$$

FIGURE 3.4: Reduceron machine description.

Before we start describing the transition rules, in Figure 3.5 we describe some auxiliary functions used by the transition rules. The function $arity$ gets the arity of the atoms that can occur at the top of the stack. The atom arity of $INT\ i$ is 1, the atom $PRI\ s$ expects 2 arguments, the atom $FUN\ n\ i$ has arity n, and finally the atom $CON\ n\ i$ has arity $n + 1$, which is the number of arguments plus a case table argument. The $applyPrim$ rule applies a given primitive to its 2 arguments, in case of arithmetic primitive it returns an $INT\ i$ with the result i, or if the primitive is logic then it gives back a constructor $CON\ 0\ 0$ for $false$ or $CON\ 0\ 1$ for $true$ value. The rule $instApp$ instantiates the function body, it gives back the replacement of the formal parameters with arguments from the reduction stack and the relative pointer addresses are turned into absolute addresses. The

$$
\begin{array}{llll}
arity & (INT\ i) & = & 1 \\
arity & (PRI\ p) & = & 2 \\
arity & (FUN\ n\ i) & = & n \\
arity & (CON\ n\ i) & = & n+1
\end{array}
$$

$$
\begin{array}{lll}
applyPrim & \text{“(+)”}\ n\ m & = INT\ (n+m) \\
applyPrim & \text{“(−)”}\ n\ m & = INT\ (n-m) \\
applyPrim & \text{“(<=)”}\ n\ m & = bool\ (n <= m) \\
applyPrim & \text{“(==)”}\ n\ m & = bool\ (n == m) \\
applyPrim & \text{“(/ =)”}\ n\ m & = bool\ (n/=m)
\end{array}
$$

$$
\begin{array}{lll}
bool & False & = CON\ 0\ 0 \\
bool & True & = CON\ 0\ 1
\end{array}
$$

$$
\begin{array}{lll}
instApp\ s\ h & = map\ (inst\ s\ (length\ h)) & \\
& \textbf{where} & \\
& inst\ s\ base\ (PTR\ p) & = PTR\ (base+p) \\
& inst\ s\ base\ (ARG\ i) & = s_i \\
& inst\ s\ base\ a & = a
\end{array}
$$

$$
updateHeap\ i\ ap\ h \quad = take\ i\ h\ +\!\!+\ [ap]\ +\!\!+\ drop\ (i+1)
$$

FIGURE 3.5: Auxiliary functions used by Reduceron transition rules.

last auxiliary function *updateHeap* modifies the heap h with the application ap at heap address i.

3.4.1 Transition rules for Reduceron

The transition rules [1] are defined in Figures 3.6 and 3.7. There is one transition rule for each *Atom* in Figure 3.1.

- The transition rule *PTR* is used when the top of the stack is a pointer x to an application. The evaluation proceeds by unwinding (The application is copied from the heap to the stack). A pair is pushed onto the update stack: the heap address x, and the size of the reduction stack increased by one *1 + length s*.

- The rule for integer literals INT swaps the top of the stack and the subsequent atom. The integer arguments of a primitive application must be fully evaluated before the application can be reduced. At run time because an integer literal is already reduced, the subsequent atom needs to be reduced by the rule $n\ e \rightarrow e\ n$, where n is an integer literal and e is another expression.

- The rule for primitive functions PRI assume that the integer arguments are fully evaluated before the application is reduced. At compile time the primitive applications are transformed by the rule $prim\ e_0\ e_1 \rightarrow e_1\ (e_0\ prim)$, by doing this we can first evaluate its arguments (See sections 2.3 and Section 5 of [1] for further details).

- The rule for constructors CON looks for a corresponding case alternative function in case table $TAB\ i$. There is no information immediately available about the arity of the case alternative function (look at address $i + j$). However, we can use zero as arity $FUN\ 0\ (i + j) : s$, because a case alternative function is never partially applied.

- The transition rule FUN applies a function f of arity n. We need to pop $n + 1$ elements from the reduction stack. The spine application of the body of f is instantiated and pushed onto the reduction stack. Finally, the remaining function applications are instantiated and added to the heap h'.

- The transition rule *Update* checks whether the arity of the atom on top of the stack is greater than the updated stack address sa subtracted from the length of the stack plus one. If that condition holds, then the heap h' is updated $update\ ha\ (top : take\ n\ s)\ h$ in the new state.

To support the printing of the results from the computations, the Reduceron machine makes use of two primitives: In Flite, both *emit* and *emitInt* are applied to two arguments: the first is what is to be printed and the second is the result of the *emit* or *emitInt* application. The primitive *emit* is used to print a character and *emitInt* for integer numeric values. We have an extra component to our machine, which is used to store the result of the computation, we call it r. It is only used when we print or accumulate printings, for example a list of numbers from 1 to 10 will be accumulated as 12345678910. Similarly for a list of characters. We make use of two auxiliary functions which are *print* and *printC*. The function *print* shows the integer value, and *printC* the character contained in the value n as an ASCII code.

PTR rule:

$$
\begin{array}{rl}
 & < p,\ \ h,\ \ PTR\ x : s,\ \ u\ \ > \\
\Longrightarrow & < p,\ \ h,\ \ h_x : s,\ \ upd : u\ \ > \\
 & \textbf{where} \\
 & upd = (1 + length\ s{,}x)
\end{array}
$$

INT rule:

$$
\begin{array}{rl}
 & < p,\ \ h,\ \ INT\ n : x : s,\ \ u\ \ > \\
\Longrightarrow & < p,\ \ h,\ \ x : INT\ n : s,\ \ u\ \ >
\end{array}
$$

PRI rule:

$$
\begin{array}{rl}
 & < p,\ \ h,\ \ PRI\ f : x : y : s,\ \ u\ \ > \\
\Longrightarrow & < p,\ \ h,\ \ applyPrim\ f\ x\ y : s,\ \ u\ \ >
\end{array}
$$

CON rule:

$$
\begin{array}{rl}
 & < p,\ h,\ CON\ n\ j : s,\ u > \\
\Longrightarrow & < p,\ h,\ FUN\ 0\ (i + j) : s,\ u > \\
 & \textbf{where} \\
 & TAB\ i = s_n
\end{array}
$$

FUN rule:

$$
\begin{array}{rl}
 & < p,\ h,\ FUN\ n\ f : s,\ u > \\
\Longrightarrow & < p,\ h',\ s',\ u > \\
 & \textbf{where} \\
 & (pop, spine, apps) = p_f \\
 & h' = h\ \ +\!\!+\ \ map\ \ (instApp\ \ s\ \ h)\ \ apps \\
 & s' = instApp\ \ s\ \ h\ \ spine\ \ +\!\!+\ \ drop\ \ pop\ \ s
\end{array}
$$

Update rule:

$$
\begin{array}{rl}
 & < p,\ h,\ top : s,\ (sa, ha) : u > \\
\Longrightarrow & state \\
 & \textbf{where} \\
 & n\ \ = 1 + length\ \ \ s - sa \\
 & h'\ \ = updateHeap\ \ \ ha\ \ (top : take\ \ n\ \ s)\ \ \ h \\
 & state\ \ = < p\ ,\ \ h',\ \ top : s,\ \ u > \ \textbf{if}\ \ arity\ top > n
\end{array}
$$

FIGURE 3.6: Reduction rules for Reduceron.

PRI emitInt rule:
$$< p,\ h, PRI\ "emitInt" : e : s,\ u,\ r > \\ \Longrightarrow\ < p,\ h,\ e : PRI\ "emitInt" : s,\ u,\ r >$$

PRI emit rule:
$$< p,\ h,\ PRI\ "emit" : e : s,\ u,\ r > \\ \Longrightarrow\ < p,\ h,\ e : PRI\ "emit" : s,\ u,\ r >$$

INT emitInt rule:
$$< p,\ h,\ INT\ n : PRI\ "emitInt" : s,\ u,\ r > \\ \Longrightarrow\ < p,\ h,\ s,\ u,\ r +\!\!+ print\ n >$$

INT emit rule:
$$< p,\ h,\ INT\ n : PRI\ "emit" : s,\ u,\ r > \\ \Longrightarrow\ < p,\ h,\ s,\ u,\ r +\!\!+ printC\ n >$$

FIGURE 3.7: Reduction rules for Reduceron(Emit and EmitInt Rules).

- The rules $PRI\ emitInt$ and $PRI\ emit$ swap the top of the stack and the subsequent atom denoted by e, by doing this, the expression e will be evaluated first.
- The $INT\ emitInt$ takes out the top two atoms from the stack, and accumulates n from $INT\ n$ as a printed result in r.
- The $INT\ emit$ takes out the top two atoms from the stack, and accumulates the character value n from $INT\ n$ as a printed result in r.

3.4.2 An Example: Minimum of Two Values

The state is a tuple $(Program \times Heap \times Stack \times UStack)$. The program minim in Figure 3.8 is executed by following the transition states in Figure 3.9, it illustrates how the Reduceron rules are applied. The program minim calculates the minimum value of two given values. The initial state is $(p, [\,], [FUN\ 0\ 0], [\,])$, here p is the program minim. The final state is when the stack is $[INT\ i]$. In this example the last state is $(p, [[PRI\ (<=), INT\ 10], [\,]], [INT\ 1], [\,])$.

```
main = minim 1 10
minim a b = case ((<=) a  b) of {
            False -> b;
            True  -> a;
            }
```

The case expressions are translated to case table

```
main = minim 1 10
minim a b         = (b (a (<=))) <minimFalse,minimTrue>
minimFalse  a b t = b
minimTrue   a b t = a
```

Translation to the actual template code requires positional reference to templates, arguments and a single off-spine application. Recall that the first component in each template is its arity.

```
(0,[FUN 2 1,INT 1,INT 10],[])
(2,[ARG 1,PTR 0,TAB 2],[[(ARG 0,PRI (<=)]])
(3,[ARG 1],[])
(3,[ARG 0],[])
```

FIGURE 3.8: An illustrative example of *minim* program.

initial state
$< p, [\,], [FUN\ 0\ 0], [\,] >$

$\Rightarrow$ (**FUN**)
$< p, [\,], [FUN\ 2\ 1, INT\ 10, INT\ 1], [\,] >$

$\Rightarrow$ (**FUN**)
$< p, [[INT\ 10, PRI(<=)]], [INT\ 1, PTR\ 0, TAB\ 2, INT\ 1, INT\ 10], [\,] >$

$\Rightarrow$ (**INT**)
$< p, [[INT\ 10, PRI(<=)]], [PTR\ 0, INT\ 1, TAB\ 2, INT\ 1, INT\ 10], [\,] >$

$\Rightarrow$ (**PTR**)
$< p, [[INT\ 10, PRI(<=)]], [INT\ 10, PRI(<=), INT\ 1, TAB\ 2, INT\ 1, INT\ 10], [(5, 0)] >$

$\Rightarrow$ (**INT**)
$< p, [[INT\ 10, PRI(<=)]], [PRI(<=), INT\ 10, INT\ 1, TAB\ 2, INT\ 1, INT\ 10], [(5, 0)] >$

$\Rightarrow$ (**Update**)
$< p, [[PRI(<=), INT\ 10]], [PRI(<=), INT\ 10, INT\ 1, TAB\ 2, INT\ 1, INT\ 10], [\,] >$

$\Rightarrow$ (**PRI**)
$< p, [[PRI(<=), INT\ 10]], [CON\ 0\ 0, TAB\ 2, INT\ 1, INT\ 10], [\,] >$

$\Rightarrow$ (**CON**)
$< p, [[PRI(<=), INT\ 10]], [FUN\ 0\ 2, TAB\ 2, INT\ 1, INT\ 10], [\,] >$

$\Rightarrow$ (**FUN**)
$< p, [[PRI(<=), INT\ 10], [\,]], [INT\ 1], [\,] >$

FIGURE 3.9: An illustrative example of transitions in Reduceron machine.

$$
\begin{array}{lcl}
Description & ::= & String \\
Id & ::= & Int \\
Failure & ::= & Id \times Description \\
SF & ::= & State \oplus Failure
\end{array}
$$

FIGURE 3.10: RunCheck transition function.

3.5 RunCheck : The Dynamic Checking Model

The model RunCheck is a *totalized* model of the operational semantics of Reduceron described in §3.4. With this model we intend to detect and classify the errors arising during the execution.

In Figure 3.10 RunCheck is a transition function from state to a new state *or a trapped error*, $S \rightarrow SF$. Where the Id is the number of the failure, and the *Description* is a string telling a brief information about the failure.

The sum type of the result SF captures the idea of a dual behaviour. When the machine *fails* the transition yields a tuple of id and error description $Id \times Description$. If the machine can perform the computation from one state to another, then it returns a valid state.

The totalized transition rules are defined in Figure 3.11. Only the rules for PTR, PRI, FUN, CON, and $Update$ are changed in this evaluation model. In addition, we changed the *Arity* auxiliary function when the element on the top of the stack is not defined.

- In the transition rule PTR the pointer x must be a reference to a valid application in the heap. We capture this idea in the constraint $0 <= x <$ length h, if this condition fails then we return an invalid heap address error.
- In the rule for primitive functions PRI, we add the condition to check if a primitive is in the set of valid primitives ps= {(+),(<=),(−),(==),(/ =)}.
- In order to totalize the rule CON, we need to add the condition $n <$ length s and $s\ !\ n = TAB\ i$, which means that the arguments are valid in the scope of the stack, and that the index of the constructor i points to a $TAB\ i$ constructor, otherwise the rule returns a bad stack pattern error.

- The transition rule *FUN* applies a function f of arity n. We need to pop $n + 1$ elements from the reduction stack. The spine application of the body of f is instantiated and pushed onto the reduction stack. Finally, the remaining function applications are instantiated and added to the heap *h'*.
- The transition rule *Update* the condition that must hold, is the arity checking $arity\ top > n$. In the auxiliary rule *Arity*, we have decided to assign a negative arity in case that the atom in the top of the stack is not a valid atom.

3.6 Mutating Reduceron Code

In our experiments to evaluate code-checkers, we need a suitable framework to emulate the situation when the code is altered or modified, perhaps maliciously to cause some damage during the evaluation in the target machine. Possible approaches include : randomly created programs, hand-coded programs, and mutated programs. Each has advantages and drawbacks.

Randomly created programs Let us firstly consider the idea of producing arbitrary random programs [75–78]. It seems to fit in our framework for testing. The framework *QuickCheck* [77] could be used to produce programs randomly. However the random approach is not a good candidate for our testing purposes because the programs created are completely *arbitrary*. For instance, we could have a empty list [] representing a list of templates. The design for obtaining programs close to genuine programs or programs that are not too expensive in terms of computation steps is complex in the random scenario.

Alteration by hand Another possibility is to create programs completely by hand, or by hand modification. This provides a good technique and close to what we want: bad code produced by hand or altered by hand. According to some experiments of hand-writing code we realise that it is easy to make a mistake, for example confusion in the order of function arguments. The drawback of this technique is that it takes too much time to produce programs compared to randomly created programs for example.

PTR rule

$$
\begin{array}{ll}
 & < p,\ h,\ PTR\ x : s,\ u > \\
\Longrightarrow & state \\
 & \textbf{where} \\
 & upd = (1 + length\ s,\ x) \\
 & state = < p,\ h,\ h_x : s,\ upd : u > \qquad \textbf{, if } 0 <= x < length\ h \\
 & state = (2,\text{'Invalid HEAP address'}) \qquad \textbf{, otherwise}
\end{array}
$$

PRI rule

$$
\begin{array}{ll}
 & < p,\ h,\ PRI\ f : x : y : s,\ u > \\
\Longrightarrow & state \\
 & \textbf{where} \\
 & ps = \{(+),(<=),(-),(==),(/=)\} \\
 & state = < p,\ h,\ primApply\ f\ x\ y : s,\ u > \qquad \textbf{, if } f \in ps \\
 & state = (7,\text{'Invalid primitive'}) \qquad \textbf{, otherwise}
\end{array}
$$

CON rule

$$
\begin{array}{ll}
 & < p,\ h,\ CON\ n\ j : s,\ u > \\
\Longrightarrow & state \\
 & \textbf{where} \\
 & state = < p,\ h,\ FUN\ 0\ (i + j) : s,\ u > \qquad \textbf{, if } n < length\ s \wedge s_n = TAB\ i \\
 & state = (5,\text{'Bad STACK pattern'}) \qquad \textbf{, otherwise}
\end{array}
$$

FUN rule

$$
\begin{array}{ll}
 & < p, h, FUN\ n\ f : s,\ u > \\
\Longrightarrow & state \\
 & \textbf{where} \\
 & (pop, spine, apps) = p_f \\
 & h' = h\ +\!\!+\ map\ \ (instApp\ \ s\ \ h)\ \ apps \\
 & s' = instApp\ \ s\ \ h\ \ spine\ +\!\!+\ drop\ \ pop\ \ s \\
 & state = < p,\ h',\ s',\ u > \qquad \textbf{, if } f < length\ p \\
 & state = (5,\text{'Bad Template address'}) \qquad \textbf{, otherwise}
\end{array}
$$

Update rule

$$
\begin{array}{ll}
 & < p,\ h,\ top : s,\ (sa, ha) : u > \\
\Longrightarrow & state \\
 & \textbf{where} \\
 & n = 1 + length\quad s - sa \\
 & h' = update\quad ha\quad (top : take\ n\ s)\ h \\
 & state = < p,\ h',\ top : s,\ u > \qquad \textbf{, if } arity\ top > n \\
 & state = (5,\text{'Bad STACK pattern'}) \qquad \textbf{, otherwise}
\end{array}
$$

FIGURE 3.11: Reduction rules for RunCheck.

Mutated programs The mutation technique was proposed by DeMillo [79] late in the 70's, and followed by [80–87]. By mutating we intend to emulate the scenario in which we have code close to genuine code. This mutated code is the original code *slightly modified*. The idea of mutation is to start from an original valid program (that terminates under the operational semantics).

Once we have this *genuine* program, it is mutated by altering selected atoms, or changing the position of two contiguous atoms, or deleting an arbitrary atom. By performing small mutations, we create a *mutant* based on a genuine program and not arbitrary programs as in the random approach.

The number of mutations that we can create is in proportion to the number of atoms in the original program. Mutation takes the best of the other two techniques: realistic programs produced automatically. If the number of atoms in a given program is too small to produce enough mutations, we can create compound mutations based on a series of mutations (depth zero,depth 1,...,depth n). In the literature of mutation testing the mutations of depth one are called *FOM* (first order mutations) and the ones of any depth greater than one are called *HOM* (higher order mutation) [33].

Mutated programs and Random Selection For the purpose of our experiments, we generate first a list of all the possible mutations, and secondly, from that list we *randomly* extract an arbitrary number of mutations.

Our Mutation Testing Approach. For our work we reuse the idea of *damaging code* from mutation testing. But the use we make of mutant programs is different. In classic mutation testing, mutations are used to measure the effectiveness of a test suite in terms of the ability to detect faults [33]. If a test suite can detect a problem in a mutant, then that mutant is "killed" (a good thing!). A key step in mutation testing is to compare the output of the original valid program against the output of each mutant.

In our case we do not compare results computed by each mutation with those computed by the original program. Instead for each mutant we compare its behaviour under the operational semantics, against the outcome of static checking. Our aim is to measure the effectiveness of the static checker. We do not care about

the correctness of a mutant; if the machine computes any value, without crashing, that is fine for our purposes.

The well known problem of *equivalent mutants* [33] can arise when mutation testing is used to evaluate test suites. A mutant may happen always to compute the same result as the original program, so no test suite can kill it. This problem is no concern here. We only want to know if each mutant is well-behaved or not, and if it is well-checked or not. The result it computes is ignored.

Random Testing and Further Discussion Another way of constructing realistic test programs is the use of random generators based on with attribute-grammars, as in the work of Drienyovszky, Horpasci et al. [88], where QuickCheck is applied to test refactoring tools for Erlang. The attribute-grammar generator provides a more expressive mechanism [89] than a context-free-grammar generator. The code for the generators is more concise and maintainable than one developed using the standard generator method of QuickCheck.

The idea of using program *generators* to automate the testing of programming-language tools can be quite successful. For instance, in the work of Daniel, Dig et al. [90] they found 45 previously unreported bugs in Eclipse and NetBeans, which are the most used refactoring tools for Java. Here the idea of the generators is taken from QuickCheck generators, even if they do not use random testing. This approach uses Java classes and an imperative and exhaustive way of generating test programs.

We could have used similar program-generation techniques to test our tools. However, we decided to use the idea of mutants because it is a simple approach to implement, and it produces tests that are quite close to a well-behaved program.

3.7 Kinds of Mutations

3.7.1 Introduction

Recall that a template body includes spine-application and off-spine applications. Every application is a sequence of atoms. The idea of mutation in our framework is to *damage* a single atom. Even such minimal damage can be enough to change

completely the behaviour of a genuine piece of code. The algorithm to mutate the template traverses the structure of the template, then by applying a simple rule over atoms we obtain a new program which we call a *mutant*. We mutate a single atom and obtain one mutation.

Increment Mutations To explain this idea consider an atom *FUN i j* which denotes a function with arity i and index j. The main idea is to attack slightly the code by atomic mutation. We can alter i or j by *incrementing*. For instance if $j = 1$ we then can increment by one and obtain a *mutated atom*. For example the original *FUN 1 1* atom will become *FUN 1 2*, with this slight change we have obtained a completely new function.

Now suppose that, in another template code, we have a function argument denoted by *ARG 2*. If we mutate it to *ARG 1* or *ARG 3*, the meaning of applying one given function to the mutated argument is completely different. We see, in this way we can alter every single numeric value inside each constructor, we call this type of mutation *increment* mutation.

Deletion of Atoms Another way of mutating the code is the *deletion* of one arbitrary atom in the template, for instance in the application *[FUN 2 2, ARG 0, ARG 1]*, we can delete *ARG 0* or *ARG 1*, letting the function application have only one argument instead of two. Or we can delete the atom *FUN 2 2* as a more radical mutation.

Swapping of Atoms The third type of mutation is to *swap* two adjacent atoms. From our example *[FUN 2 2,ARG 0,ARG 1]* we can get *[ARG 0,FUN 2 2,ARG 1]* or *[FUN 2 2,ARG 1,ARG 0]*. This last small mutation evokes the common error when a programmer changes the order of the arguments in one function.

Even the single atomic mutations can damage or alter the code significantly. For the human reader it is almost impossible to distinguish between original valid code and mutated variants. Imagine pages of template code: such small changes are imperceptible. The mutated code can be very close to the original but it can be wrong. The three types of mutations can be combined causing possibly more damage.

```
length=  [(0,[FUN 1 1,PTR 0],[[CON 0 1]]),
          (1,[ARG 0,TAB 2 2 0],[]),
          (3,[FUN 1 1,ARG 1,PTR 0],
             [[INT 1,PRI "(+)"]]),
          (1,[INT 0],[])]
```

FIGURE 3.12: Template code for *length* function

3.7.2 Mutation Rules

Increment Mutations In general, the number of mutations from a given program p is directly proportional to the number of atoms and the number of numerical values in the atomic constructors. For example in Figure 3.12, if we want to use the increment mutation, the atoms excluding the first template (it is the *main* function) are *[ARG 0, TAB 2 2 0 , FUN 1 1, ARG 1, PTR 0, INT 1, PRI "(+)", INT 0]*.

From the rule for mutating atoms in Figure 3.14, we see that the mutations depend on the rule *[mInt]* which mutates the numeric values from *n+1 to n+3*, in Table 3.1 we have evidence of the rounded average in some numeric values. In general the average of those numeric values is 2. Based on that observation we decided to mutate from $n+1$ to $n+3$. Using more than $n+3$ will create non-useful mutations. For instance in the *Template Arity* column in Table 3.1 we can see that the possible number of arguments is on average 2. Then mutating the *ARG 0* we obtain *ARG 1*, *ARG 2*, *ARG 3*, using the convention $n+1$ to $n+3$.

And the *[mPri]* rule which selects a different primitive from the five possible primitives.

If we apply the *mA* rule to each atom in the length function, we obtain 28 mutations, and they are distributed for each atom in Figure 3.13. Note the rule for mutate the atom *TAB i j k*, for our experiments we have mutated only the numeric value i, which represents the address position of the first case alternative. We leave the other two constructors unchanged, is because in the operational semantics the arguments j and k are not considered during the reductions.

For our experiments we generate all the possible mutations, then we select n mutations randomly. The advantage of this approach is that we can test the mutations randomly or take all the possible mutations generated. Moreover, we can mutate mutations, which is useful when we have small programs to test. For

```
ARG 0       3 mutations , TAB 2 2 0   3 mutations
FUN 1 1     6 "         , ARG 1       3 "
PTR 0       3 "         , INT 1       3 "
PRI "(+)"   4 "         , INT 0       3 "
```

FIGURE 3.13: Number of mutations for each atom in *length* function

instance, the mutation of function *length* only produces 28 mutations. So possibly under this case it is better to mutate the 28 mutations.

Swapping Mutations This rule swaps two contiguous elements in a list, and creates all swapped lists excluding the original list.

```
swaps xs= nub([i ++ swap t|(i,t) <- zip (inits xs) (tails xs)])\\[xs]

swap []       = []
swap (y:x:xs) = (x:y:xs)
swap x        = x
```

Delete Mutations This rule deletes one element in a list at a time, and creates all the deleted lists. It deletes only if there is more than one atom in the list of atoms denoted by *xs* to avoid the "obvious" generation of a bad template (arity,[],[]).

```
deletesp xs= [delete (xs!!i) xs| i<-[0..length xs-1], length xs > 1]
```

All Mutations The kind of mutation that combines the previous mutations is called All, and it is given by Increment + Swapping + Delete.

The mutator could be extended easily to support other types of mutations, for example swapping or deleting applications.

$$mA\ [\![FUN\,n\,i]\!] \Rightarrow \{FUN\,n'\,i \mid n' \in\ mInt\ n\} \cup \{FUN\,n\,i' \mid i' \in\ mInt\ i\}$$

$$mA\ [\![ARG\,m]\!] \Rightarrow \{ARG\,m' \mid m' \in\ mInt\ m\}$$

$$mA\ [\![PTR\,i]\!] \Rightarrow \{PTR\,i' \mid i' \in\ mInt\ i\}$$

$$mA\ [\![CON\,n\,m]\!] \Rightarrow \{FUN\,n'\,m \mid n' \in\ mInt\ n\} \cup \{CON\,n\,m' \mid m' \in\ mInt\ m\}$$

$$mA\ [\![INT\,i]\!] \Rightarrow \{INT\,i' \mid i' \in\ mInt\ i\}$$

$$mA\ [\![PRI\,s]\!] \Rightarrow \{PRI\,s' \mid s' \in\ mPri\ s\}$$

$$mA\ [\![TAB\,i\,j\,k]\!] \Rightarrow \{TAB\,i'\,j\,k \mid i' \in\ mInt\ i\}$$

$$mInt\ [\![n]\!] \Rightarrow \{n_{+1}..n_{+3}\}$$
$$mPri\ [\![s]\!] \Rightarrow \{+, -, ==, <=, /=\} - \{s\}$$

FIGURE 3.14: Rule to mutate atoms by incremental damage.

TABLE 3.1: Averages of templates arities, case alternatives arities, and function arities in the set of benchmark programs.

Program	Template Arity	Tab Alts	Fun Arity
queens	2	2	2
queens2	2	2	2
ordlist	2	2	2
braun	2	2	2
while	3	2	2
adjoxo	2	2	2
sudoku2	2	2	2
mss	2	2	2
permsort	2	2	2
parts	2	2	2
taut	2	2	2
clausify	2	2	2
cichelli2	2	2	2
mate	3	2	2
knuthbendix	3	2	2

3.8 Classification of Results

We have created a classification to measure the effectiveness of our static checking techniques for Reduceron code. The Venn diagram in Figure 3.15 depicts this classification.

The *Mu* set represents a list of mutations chosen randomly. Here it is important to notice that we only use the *needed templates* to generate mutations. We run a simple analysis to detect which templates are used during the execution. That information is passed to the mutator at mutation time. This checking is fundamental for our experiments, some of the benchmark programs take too long to evaluate, and we use the minimal initial configuration to have the most possible templates executed in the least execution time. Consider the case where we have 200 mutations and each mutation takes one minute to be evaluated, then it will take more than three hours to compare the run-time outcome against the static technique.

Also, we eliminate the set of dead code *Dc*, which is not present in the diagram. We execute the original program and determine which templates are used. After this analysis, we only mutate those templates used during the execution.

The set *Wf* is the set of well-formed mutations. By first applying a simple well-formedness test to mutants we eliminate the obvious failures, e.g. a pointer out of the scope or a primitive wrongly called "(++)". In Chapter 4 we will describe the well-formed analysis which is called AtomCheck.

The *Nt* set represents the set of programs with non-termination issues. Even if the original test program might terminate, mutated variants may not. We make use of a conservative test, to approximate the conditions under which a program does not terminate. If the number of steps to execute the mutated program exceeds double the number of steps needed to execute the original program, the mutant is assumed to be non-terminating.

The set *Ok* is the set of programs that are well-behaved in the Reduceron machine. We assume that they have passed the non-termination test described before.

The set *Sc* is the set of programs well-checked in some static method. In the development of this thesis, we have several stages of static checking.

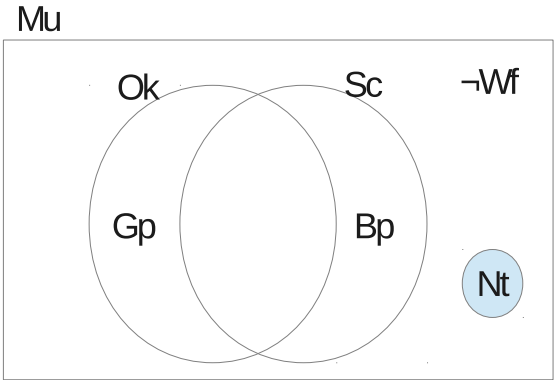


FIGURE 3.15: Mutation classification diagram.

The set Gp represents the programs which are ill-checked yet well-behaved: these programs are rejected by the static checker, but if executed they would not in fact cause a machine failure by the run-time system.

Finally, the Bp set is the set of programs which are allowed by the static checker and stopped by the run-time system.

3.9 Summary

The Reduceron uses the template instantiation machine model, but has parallel circuits to create the instance, update the stack, and reduce the graph in one single clock cycle. We have "totalized" the operational semantics for Reduceron to be able to detect and classify possible faults. The tool kit for testing and experimenting is based on a mutation technique where the mutations are chosen randomly. In this way we have test programs that are very close to the original and well-behaved code. Moreover, by repeating mutation if necessary, a large number of programs to test can be generated automatically.

Using mutated code variants as a test, we can evaluate the static methods against the operational semantics in order to measure the effectiveness of the static techniques proposed in this work.

Chapter 4

TyreCheck

4.1 Introduction

In this chapter we present an *equation-based* type-checking system for Reduceron *template code*. Even when *no extra annotations* are added to the template code we can compute type information from the raw template code. The purpose of this type-checking system *TyreCheck* – Type-based Reduceron Code Checker –, is to guarantee a form of type safety at loading time before executing the template code. As outlined in Chapter 3 we use *mutated* programs to measure the effectiveness of *TyreCheck* experimentally.

Static checking of the template code is divided into three stages as follows :

- AtomCheck. In §4.2 we describe this stage. This stage checks that each atom of the template code satisfies requirements for well-formedness.
- PrimCheck. Static analysis that eliminates primitive applications, avoiding the need for dependent types. This part is described in §4.3.
- TyreCheck. Type reconstruction based on equations instead of the traditional type inference rules. The details of this static method are described in §4.4. This is the most complex part of the three.

In §4.5 we present some results from an evaluation of TyreCheck effectiveness. Finally, in §4.6 we give a brief discussion of our techniques.

```
type Program  = [Template]

type Template = (Arity,App,[App])

type Arity    = Int

type App      = [Atom]


data Atom     = INT Int
              | FUN Arity Int
              | ARG Int
              | PRI String
              | CON Arity Int
              | PTR Int
              | TAB Int Int Int
```

FIGURE 4.1: Haskell data type representation of Template code.

From TAB i to TAB i j k During development of an initial static checker a need became apparent to know how many alternatives belong to each case table, and also to determine how many variables are in use. As a cheap method to provide this information we decided to add two parameters to TAB atoms. Atom TAB i becomes TAB i j k, where j is the number of case alternatives associated with that TAB, and k is the number of free variables in all the case alternatives. We made this change at compile time, so no extra time is required during the static analysis. At compile time we know the information about the number of case alternatives and the number of free variables. This small change is not comparable to the change of adding type signatures or logic-based formulae as code annotations.

4.2 AtomCheck

The template code can be represented by a Haskell data type as in Figure 4.1. A program is a list of templates. Each template represents a function definition consisting of arity, spine application, and off-spine applications. An application is a list of atoms, which is either a numerical literal represented by INT Int, or a function with the arity and the index of the function in the list of templates represented as FUN Arity Int, or a function argument ARG Int, or a numeric or boolean primitive PRI String, or a pointer to an application PTR Int, or a special case table atom used to encode case expressions and represented as TAB Int Int Int, where

the first parameter is the index of the first template to be considered as a case alternative, the second argument is the number of alternatives used by that case table, and the third argument is the number of free variables in the case table. In the rest of this Chapter, we will refer to this Haskell data type as the template code which we are checking.

The first method in the process checks that template code is *free of ill-formed atoms*. For example we reject a primitive atom PRI "(++)" which is not part of the valid primitive set, or a function FUN i j, where j is an index greater than the number of the actual templates in the code.

In Figure 4.2 we present a model for checking well-formedness at the atomic level. The rule *cProg* is applied to each template T in the program P. The rule *cTemp* examines every single application inside of a template. For each atom in each application it applies the *cAtom* rule.

For each template T $\in$ P:

- For each atom FUN n a the address a must be greater than or equal to zero, and less than or equal to the number of templates in the program.
- For each atom ARG i the index i of the argument must not exceed the size of the arity of the enclosing template.
- For each atom PTR i the off-spine application index i must be greater than or equal to zero, and less than the number of off-spine applications in the enclosing template.
- For each atom CON n i the arity n and the index i must both be greater than or equal to zero.
- For each atom PRI s, the primitive name s must belong to the set of valid primitives.
- Finally, for each atom TAB i j k each of the indexes i, j must both be greater than zero, and k must be non-negative. In addition the sum of i and j, must be less than or equal to the number of templates in the program.

$$cProg \llbracket \text{ P } \rrbracket \Rightarrow \forall T \in P : \quad cTemp \llbracket \text{ T } \rrbracket$$

$$cTemp \llbracket \text{ arity, spineApp, offspineApps } \rrbracket \Rightarrow \forall\, a \in spineApp \cup offspineApps \; : \; cAtom \llbracket \; a \; \rrbracket$$

$$\begin{array}{lcl}
cAtom \llbracket \text{ FUN n addr } \rrbracket & \Rightarrow & addr \geq 0 \wedge addr \leq lp \\
cAtom \llbracket \text{ ARG m } \rrbracket & \Rightarrow & arity > m \\
cAtom \llbracket \text{ PTR addr } \rrbracket & \Rightarrow & addr \geq 0 \wedge addr < la \\
cAtom \llbracket \text{ CON n m } \rrbracket & \Rightarrow & n \geq 0 \wedge m \geq 0 \\
cAtom \llbracket \text{ INT i } \rrbracket & \Rightarrow & True \\
cAtom \llbracket \text{ PRI s } \rrbracket & \Rightarrow & s \in \{(+),(-),(==),(/=),(<=),\text{emit},\text{emitInt}\} \\
cAtom \llbracket \text{ TAB i j k } \rrbracket & \Rightarrow & i > 0 \;\wedge\; j > 0 \;\wedge\; k \geq 0 \;\wedge\; i + j \leq lp
\end{array}$$

where

$arity$	= arity of the enclosing template
lp	= # of the enclosing program
la	= # of enclosing application

FIGURE 4.2: AtomCheck conditions of well-formed atom.

4.2.1 Measuring Effectiveness of AtomCheck

In Table 4.1 we see the results of evaluating AtomCheck alone against the operational semantics. The experiments of table are based on 200 mutations chosen randomly. For example, from the 200 mutations in `queens` we have 48 well-behaved programs, 171 accepted by AtomCheck, zero good programs rejected by AtomCheck, and 111 bad programs accepted by AtomCheck. For the rest of the programs similar proportions of the results are observed. One notable part of this table is column `GP`, which represents the well-behaved programs stopped by AtomCheck. AtomCheck is not rejecting valid programs.

4.3 PrimCheck

In the next stage we analyse and transform template code to *reduce* and *isolate* the problems derived from uses of primitive operations. The high-level representation of $prim\ a\ b$, where a primitive function $prim$ is applied to the arguments a and b, is compiled to template code applications equivalent to $b\ (a\ prim)$. This order allows

Table 4.1: Mutations (200-All-AtomCheck)

Program	OK	AtomCheck	GP	BP	NT
queens	48	171	0	111	12
permsort	41	164	0	120	3
queens2	51	167	0	113	3
ordlist	55	168	0	110	3
parts	53	170	0	112	5
braun	31	162	0	125	6
mss	26	170	0	135	9
adjoxo	64	175	0	108	3
sudoku	33	167	0	130	4
tautology	45	172	0	118	9
cichelli2	37	173	0	131	5

the argument be reduced to an atom of integer type. The primitive functions must be fully saturated to satisfy the Reduceron semantics.

We have explored some alternatives, for example representing the types for primitives as dependent types indexed by arity. In addition to the implication of using dependent types in a full type system, problems arise because of the nature of partial applications during the type-checking process. Primitive applications must be *fully saturated*.

Why not Dependent Types in the Type-checking System? The compilation rule for primitives is $prim\ a\ b \rightarrow b(a\ prim)$. From the point of view of types this arrangement where the arguments are applied to the primitives represents a challenge. In a standard type system, if we have an arithmetic primitive with two arguments, its type is $int \rightarrow int \rightarrow int$. If we are to keep an applicative view of Reduceron code, what types can be assigned to the primitive arguments (a and b above) used as functions? One possible solution to address the problem is to use dependent types: we might introduce type equivalences for integers $int = prim\ (n+1)\ r \rightarrow prim\ n\ r$ and $int = prim\ 0\ r$.

For example, suppose we have the template

(2, [ARG 1,PTR 0], [[ARG 0,PRI(+)]]). In the application [ARG 0,PRI(+)] the type of ARG 0 is $prim\ 2\ r \rightarrow prim\ 1\ r$, a functional type denoting the reduction in the arity from two to one, but preserving the result type denoted by r. We need

to distinguish between arithmetic and logic primitives, as r could be either *int* or *bool*. Binary primitives have type *prim* 2 r, the type of PRI(+) is *prim* 2 *int*.

Now for the application of the argument ARG 1 to [ARG 0,PRI (+)] we apply the same mechanism. The type for the whole template body

[ARG 1,PTR 0],[[ARG 0,PRI(+)]] is *prim* 0 r. We identify the type prim 0 r with r.

Trying to encode all these possibilities adds more complexity to a type-checking system. Because primitives are fully saturated we even need still more testing conditions. Certainly, we could use this idea of dependent types, but it could be complicated. We have experimented with this approach but it caused too many problems. We decided to do the analysis related to primitive application in a previous stage before type-checking.

By using a primitive reduction analysis before type checking we have two gains: (1) we can detect when a primitive is not fully saturated and stop bad programs before the process of type inference, and (2) we eliminate all the primitive applications working instead with a simple integer type.

4.3.1 Divide and Conquer

Separating primitive analysis from a later stage of type-checking has some other advantages, besides the division of the work :

- We can stop some bad programs at an early stage without the full cost of type-checking.
- We avoid the need to use dependent types.

We call this stage of our static analysis *PrimCheck* for Primitive Checking. The main idea of PrimCheck is to eliminate once and for all primitive applications from the code. At the end of PrimCheck analysis, from each template in the code we have derived a *split template*, with *no* occurrences of primitives but with the possible addition of *integer blocks*. PrimCheck analysis is applied after the well-formedness test of AtomCheck. In Figure 4.3 we explain the overall process of PrimCheck.

At the implementation level, PrimCheck takes as an argument one template, and returns a pair *containing* a template and a list of *integer blocks*. In the integer blocks we obtain all the collected *arguments* or *applications that must be integers*.

All primitives in Reduceron must be applied to two arguments. The first step is to change all the occurrences of the atoms PRI s to PRIM 2 t. That is why each PRI atom can be reduced to a PRIM 2 t atom : the 2 indicates arity and the t indicates the result type of the primitive. During the process of PrimCheck arities in PRIM atom are reduced until they reach zero. The PRIM 0 t is reduced to t.

Once the replacement of the primitives is performed, we apply the *reduction rule* for primitives. It takes a pair of template and integer blocks and returns a split template. The *PrimApply* rule is applied until it reaches a fix point. The rule works as follows :

- if there are no primitives in the template, then it just finishes returning the same template.
- if there is a primitive in the head of any application it means that the primitive reduction cannot be completed, and it is a failure.
- otherwise the application in the template is split and inlined.

In order to obtain the new split application we need to check several things. There are two cases:

- If there are primitive and pointer atoms in the same application *ap*, and that pointer makes reference to any other application in the list of all the applications, and that application contains a primitive, then we return the pair (*ap*,[]), the same application and the empty list of integer blocks.
- Otherwise, we proceed to split the application *ap* by using the rule *splitPrimApp*. The heart of the algorithm is the rule *splitPrimApp*. This rule takes an application and returns an *application* and a list of *integer blocks*. The objective is to *split the application where we found a primitive*. By doing this, we obtain two lists, the first list is pulled out into an integer block. The other list is evaluated to reduce the primitive arity by one. A similar process is applied in both spine and off-spine applications.

Once we have collected all the splittings for the entire template, *inlining* is applied. The rule *inline* checks for every atom in the split template. If a PTR atom referring to a singleton application contains a singleton containing a PRIM atom, then, the primitive atom is inlined. Then the rule *soloPrims* replaces all the singleton primitive applications in the split template with empty applications. In this way, PrimCheck preserves the order of the applications in the list of the off-spine applications.

Apart from the logical and numeric primitives, there are two special primitives to print the results from the computation in the Reduceron machine. The rules to deal with atoms PRI emitInt and PRI emit are applied before the rules for the conventional primitives and they are described in rules *emitApply* and *splitEmitApp*. The main difference is in rule *splitEmitApp*, which splits the application where a *primEmit* is found. It checks which are the arguments of emit, and makes those arguments integer blocks. The application of the emit primitive is reduced to INT 0.

4.3.2 An Example Application of the PrimCheck Rules

Consider for example the template code `tri5`.

```
tri5 = [ (0, [FUN 1 1, INT 5], [])
       , (1, [INT 1, PTR 0, TAB 2 2 1,ARG 0],
             [[ARG 0, PRI "(<=)"]])
       , (2, [ARG 1, PTR 0],
             [[FUN 1 1,PTR 1,PRI "(+)"],
              [INT 1, PTR 2],
              [ARG 1, PRI "(-)"]])
       , (2, [INT 1], [])]
```

The first transformation is to replace each PRI s atom for a PRIM n b atom. When we apply the rule *introPrim* to each template in the program tri5, we obtain a list of split templates. Each split template is a pair composed by a new template with PRIMs and the empty list for integer blocks. It introduces PRIM 2 True for the logical primitives, and PRIM 2 False for the arithmetic ones, where 2 is the arity of the primitive application.

$primCheck \llbracket\ t\ \rrbracket \Rightarrow fix(primReduce(primIntro \llbracket\ t\ \rrbracket))$
where
$primReduce\ st = primApply(emitApply(st))$

$pri2prim \llbracket\ a\ \rrbracket \Rightarrow$
PRIM 2 False , **if** $a \in$ {PRI (+),PRI (-)}
PRIM 2 True , **if** $a \in$ {PRI (<=),PRI (==),PRI (/=)}
a , **otherwise**

$primIntroApp \llbracket\ a_1, ..., a_n\ \rrbracket \Rightarrow pri2prim \llbracket\ a_1\ \rrbracket, ..., pri2prim \llbracket\ a_n\ \rrbracket$

$primIntro \llbracket\ (arity, ap, ap_1, ..., ap_n)\ \rrbracket \Rightarrow ((arity, ap', aps'), [\,])$
where
$ap' = primIntroApp \llbracket\ ap\ \rrbracket$
$aps' = primIntroApp \llbracket\ ap_1\ \rrbracket, ..., primIntroApp \llbracket\ ap_n\ \rrbracket$

$primApplyApp\ \ aps \llbracket\ a_1, ..., a_n\ \rrbracket\ \ \Rightarrow$
$(a_1, ..., a_n, [\,])$, **if** $\exists i, j, p, q : isPrim\ a_i \wedge (a_j = \text{PTR p}) \wedge isPrim\ aps_{p_q}$
$splitPrimApp \llbracket\ a_1, ..., a_n\ \rrbracket$, **otherwise**

$primApply \llbracket\ st\ \rrbracket \Rightarrow$
st , **if** $\forall\ i : \neg isPrim\ a_i \wedge \forall\ i,\ j : \neg isPrim\ ap_{i_j}$
$Fail$, **if** $\exists\ i : isPrim\ ap_{i_1}\ \vee\ isPrim\ a_1$
$inline((arity, ap', ap'_1, ..., ap'_n), ints')$, **otherwise**
where
$((arity, ap, ap_1, ..., ap_n), ints) = st$
$(ap', ints_0) = primApplyApp\ \ aps \llbracket\ ap\ \rrbracket$
$(ap'_i, ints_i) = primApplyApp\ \ aps \llbracket\ ap_i\ \rrbracket, 1 <= i <= n$
$ints' = ints_0 ++ ints_1 ++ ... ++ ints_n ++ ints$

$splitPrimApp \llbracket\ a_1, ..., a_n\ \rrbracket \Rightarrow$
$(a_1, ..., a_n, [\,])$, **if** $\forall\ i : \neg isPrim\ a_i$
((PRIM (m-1) b, $a_{i+1}, ..., a_n$), $ints$) , **if** $(\forall j : 1 <= j < i, \neg isPrim\ a_j) \wedge a_i = \text{PRIM m b}$
where
$ints = [\,]$, **if** $i = 2 \wedge isInt\ a_1$
$ints = a_1, ..., a_{i-1}$, **otherwise**

FIGURE 4.3: Rules to reduce primitives in PrimCheck (I).

$$
\begin{array}{l}
inline \llbracket\ ((arity, ap, aps), ints)\ \rrbracket \Rightarrow \\
\quad nullsoloPrims((arity, ap', aps'_0, ..., aps'_n), ints) \\
\quad \textbf{where} \\
\quad ap' = inlinePtrApp\ \ aps \llbracket\ ap\ \rrbracket \\
\quad aps'_i = inlinePtrApp\ \ aps \llbracket\ aps_i\ \rrbracket, 1 <= i <= n
\end{array}
$$

$$
\begin{array}{l}
inlinePtrApp\ \ aps \llbracket\ a_1, ..., a_n\ \rrbracket \Rightarrow \\
\quad inlinePtrAtom\ \ aps \llbracket\ a_1\ \rrbracket, ..., inlinePtrAtom\ \ aps \llbracket\ a_n\ \rrbracket
\end{array}
$$

$$
\begin{array}{lll}
inlinePtrAtom\ \ aps \llbracket\ a\ \rrbracket & \Rightarrow & \\
\quad aps_i & , \textbf{if}\ a = \text{PTR i} \wedge soloPrim\ aps_i \\
\quad a & , \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
nullsoloPrims \llbracket\ ((arity, ap, aps_1, ...aps_n), ints)\ \rrbracket & \Rightarrow & \\
\quad ((arity, ap, aps'_1, ...aps'_n), ints) & & \\
\quad [\,] & , \textbf{if}\ soloPrim\ aps_i \\
\quad aps'_i & , \textbf{otherwise}
\end{array}
$$

$$
soloPrim \llbracket\ a_1 ... a_n\ \rrbracket \Rightarrow n = 1 \wedge isPrim\ a_1
$$

$$
isPrim \llbracket\ a\ \rrbracket \Rightarrow a = \text{PRIM n t}
$$

FIGURE 4.4: Rules to reduce primitives in PrimCheck (II).

$$
\begin{array}{l}
emitApply \llbracket\ st\ \rrbracket \Rightarrow ((arity, ap', ap'_1, ..., ap'_n), ints') \\
\quad \textbf{where} \\
\quad ((arity, ap, ap_1, ..., ap_n), ints) = st \\
\quad (ap', ints_0) = splitEmitApp \llbracket\ ap\ \rrbracket \\
\quad (ap'_i, ints_i) = splitEmitApp \llbracket\ ap_i\ \rrbracket, 1 <= i <= n \\
\quad ints' = ints_0 +\!\!+ ints_1 +\!\!+ ... +\!\!+ ints_n +\!\!+ ints
\end{array}
$$

$$
\begin{array}{lll}
splitEmitApp \llbracket\ a_1, ..., a_n\ \rrbracket \Rightarrow & & \\
\quad (a_1, ..., a_n, [\,]) & , \textbf{if}\ \forall\ i : \neg isEmit\ a_i \\
\quad ((\text{INT } 0, a_{i+i}, ..., a_n), ints) & , \textbf{if}\ (\forall j : 1 <= j <= i, \neg isEmit\ a_j) \\
 & \wedge\ isEmita_i \\
\quad \textbf{where} & & \\
\quad ints = [\,] & , \textbf{if}\ i = 2 \wedge isInt\ a_1 \\
\quad ints = a_1, ..., a_{i-1} & , \textbf{otherwise}
\end{array}
$$

FIGURE 4.5: Rules to reduce primitives in PrimCheck (III).

⇒ by application of *introPrim*

```
[ ((0, [FUN 1 1, INT 5], [])          , [])
, ((1, [INT 1, PTR 0, TAB 2 2 1,ARG 0],
       [[ARG 0, PRIM 2 True]])        , [])
, ((2, [ARG 1, PTR 0],
       [[FUN 1 1,PTR 1,PRIM 2 False],
        [INT 1, PTR 2],
        [ARG 1, PRIM 2 False]])       , [])
, ((2, [INT 1], [])],[]),             , [])
]
```

The next step is to see where there are primitives PRIM n b in each application. Each initial sequence of atoms before a PRIM n b in an application is extracted to become an integer block. In addition, the PRIM atom itself reduces to PRIM (n-1) b.

The *primApply* rule is applied to each split template for tri5. It first checks where the PRIM n b occurs. Using the *splitPrimApp* rule : it splits the template, separating the int block and reduces the arity n by one in the primitive.

⇒ by application of *primApply* (STAGE I)

```
[  ((0, [FUN 1 1, INT 5], [])          , [])
  , ((1, [INT 1, PTR 0, TAB 2 2 1,ARG 0],
         [[PRIM 1 True]])              , [[ARG 0]])
  , ((2, [ARG 1, PTR 0],
         [[PRIM 1 False],
          [INT 1, PTR 2],
          [PRIM 1 False]])             , [[FUN 1 1, PTR 1],[ARG 1]])
  , ((2, [INT 1], [])],[]),            , [])
]
```

Once the primitives are reduced and splitting is done, we proceed to *inline* the pointers to applications of the singleton form [PRIM n b]. The inlined applications are replaced by empty [] applications, preserving the indices of the off-spine applications.

⇒ by application of *inlining*

```
[  ((0, [FUN 1 1, INT 5], [])          , [])
 , ((1, [INT 1, PRIM 1 True, TAB 2 2 1,ARG 0],
        [[]])                          , [[ARG 0]])
 , ((2, [ARG 1, PRIM 1 False],
        [[],
        [INT 1,PRIM 1 False],
         []])                          , [[FUN 1 1, PTR 1],[ARG 1]])
 , ((2, [INT 1], [])],[]),             , [])]
```

Notice that when a candidate int block is a literal INT n, it is not appended to the integer blocks. The reason for that is that it does not make too much sense to compute the type of an int if we already know it is an int. The reduction of the arity of the primitives reaches zero :

⇒ by application of *primApply* (STAGE II)

```
[ ((0, [FUN 1 1, INT 5], [])      , [])
, ((1, [PRIM 0 True, TAB 2 2 1,ARG 0],
       [[]])                      , [[ARG 0]])
, ((2, [PRIM 0 False],
       [[],
        [PRIM 0 False],
        []])                      , [[FUN 1 1, PTR 1],[ARG 1],[ARG 1]])
, ((2, [INT 1], [])],[])          , [])]
```

When the arity of the primitives reaches zero the primitive atom is changed to one of two atoms : either a new BOOL atom for logical primitives or INT 0 for arithmetic primitives.

⇒ by application of *primApply* (STAGE II)

```
[ ((0, [FUN 1 1, INT 5], [])  , [])
, ((1, [BOOL, TAB 2 2 1,ARG 0],
        [[]])                 , [[ARG 0]])
, ((2, [INT 0],
        [[],
         [INT 0],
         []])                 , [FUN 1 1, PTR 1],[ARG 1],[ARG 1]])
, ((2, [INT 1], [])],[]),     , [])]
```

```
prop_primFree t            => all notPrimi (atoms (PrimCheck t))
  where
       atoms ((n,ap,aps),ints) = ap ++ concat (aps ++ ints)
       notPrimi (PRIM n b) = False
       notPrimi _          = True

prop_preserveApps t        => length offsApps == length offsApps'
  where
        offsApps  = getApps t
        offsApps' = getApps $ fst (fromJust $ PrimCheck t)
        getApps (a,app,apps) = apps
```

FIGURE 4.6: Some properties of PrimCheck.

As we can see, we have eliminated all the primitives from the template code and isolated the integer blocks representing their arguments. The result types of each primitive application are reflected in the BOOL and INT 0 atoms.

4.3.3 Properties of PrimCheck

Figure 4.6 shows two expected properties when we apply PrimCheck to a given template. We can test these properties, for example, by mutating the testing programs.

The first property, `prop_primFree`, is that at the end of the primitive analysis the split template must be free of any primitive atoms. The main objective of PrimCheck is to eliminate the primitive applications from the template code, so this is a natural property to check.

The other property, `prop_preserveApps`, is that the number of off-spine applications in a template is preserved. There must be the same number in the original template and the one after PrimCheck transformation. Notice that we only take into account the number of applications not their contents.

4.3.4 Measuring the Effectiveness of PrimCheck

In order to measure the effectiveness of the primitive analysis, in Table 4.2 we show some results from the testing of PrimCheck over a set of mutations of the

TABLE 4.2: Mutations (200-All-PrimCheck)

Program	AtomCheck	PrimCheck	OK	GP	BP	NT
queens	132	118	35	3	72	14
queens2	107	103	32	1	68	4
ordlist	113	112	34	0	77	1
mss	104	99	8	0	87	4
parts	102	92	21	1	63	9
braun	118	118	13	0	103	2
adjoxo	126	122	40	0	80	2
permsort	130	126	19	2	105	4
sudoku	107	105	7	0	95	3
while	146	144	22	2	117	7
tautology	113	110	15	1	94	2
cichelli2	115	106	11	0	89	6

different benchmark programs. For all of them we would like the column `GP` to be close to zero, as we do not want the PrimCheck analysis to stop well-behaved programs. One advantage of this analysis is that the programs with problems in the primitives can be stopped without using the type-checking analysis. The size of the set of `BP` at this stage is not important. After the PrimCheck is applied to the template code, the main goal of the type-checking analysis is to reduce the number of bad programs allowed (`BP`) ideally without increasing the number of good programs stopped (`GP`).

In Table 4.2 we show the results for `200` mutations of type `All`. The mutation type `All` includes `Increment + Delete + Swapping` type of mutations.

For the program `queens`, for example, we have `132` well-formed mutations. Only `35` programs evaluate successfully under the operational semantics, and `118` programs are valid under PrimCheck analysis. Notice that `GP` is three. In fact there are only two stopped good programs. Remember that the mutations are chosen randomly; in this case we have taken one mutation twice. `GP` is not `zero`, only because in the operational semantics the special rule for primitives means we can have the application in two different orders [PRI (+),ARG 0,ARG 1] or [ARG 0,ARG 1,PRI (+)], and obtain the same result. In the PrimCheck analysis we have not taken into account this situation. `BP` is quite big. At this stage of checking that is no surprise, remember that we are only dealing with problems in primitives. Finally the number of non-terminating programs in `NT` is `14`.

4.4 TyreCheck

In brief *TyreCheck* derives a collection of term-type equations from coded templates and then tries to solve them. In this section we explain first the type language and the mechanism to find a solution of a set of equations. Then in §4.4.4 we explore the rules for solving equations when we have algebraic data types. Finally, the mechanism to derive the collection of equations from the code and its formalism is described in §4.4.6.

4.4.1 Type-Terms

Before we discuss how to solve a system of equations, we need to introduce the terms that may occur as left-hand side or right-hand sides. In Figure 4.7 we have the type-term definition — we will refer to terms or types interchangeably from now on. A term may be a function $t \to t$, or a term variable denoted by v, or an integer Int.

In Reduceron code the high-level algebraic data types are compiled to constructors with arity and index. There is no explicit type information about algebraic data types, and atomically constructors from different types may be indistinguishable. We introduce in our type language a special type to encode algebraic data types, represented by $alg\ ics \oplus ext$. The arguments of alg are as follows. In the first place we a have a sequence ics of pairs $(i\ ,\vec{t})$ each representing a constructor of the type. The first component i of the pair is an `index` and the second is a list of argument types. The second argument ext represents the possible extension [91] of this algebraic data type. An extension is a pair $(v,\vec{i})$ where v is the variable which will contain the extended algebraic data type, and the list $\vec{i}$ contains the *lacks information* that represents the indexes of which alternatives are lacking in a overall algebraic data type. When we have a algebraic data type with an extension we will call it *open*.

In the case when we know the exact number of alternatives in an algebraic data type, we will make use of the type constructor $alg\ \ \overrightarrow{ics}$. This type cannot be extended and we will refer it as *closed* in the future.

$$
\begin{array}{rcl}
t & = & v \mid Int \mid t \to t \mid alg\ \overrightarrow{ics} \oplus ext \\
 & & \mid alg\ \overrightarrow{ics} \\
ics & = & i \times \vec{t} \\
ext & = & v \times \vec{i}
\end{array}
$$

FIGURE 4.7: Type-terms definition.

Extensible Types and Extension Variables The idea of *extending* algebraic data types is needed because in low-level code for constructor applications we do not have any type information that allows us to know about other constructors for the same data type. The full data type will be the sum of a type for this construction and some *extension* of this type for other constructions. We know the index of the constructor being applied and the extension must *lack* any constructor with that index.

On the other hand, when we have an application with a case-table argument, we know how many components the case subject has. Their types can be inferred from the templates for alternatives.

The role of the *extension variable* is to provide a way of referring to a possible extension of a data type. In our algorithm the extension variable is needed. For instance, to close a data type we add an equation assigning an empty algebraic data type to the extension variable.

4.4.2 Colmerauer's Method to Solve Recursive Equations

Our solver is based on the rules for solving recursive equations between simple applicative terms, given by Colmerauer [19] in his paper titled "Prolog and Infinite Trees". That means that we can work with recursive functions or recursive data types. All we need is to pass that *recursive* information encoded in the system of equations to be solved. However, the solver from Colmerauer only presents simple term constructors. In our case we need a mechanism to extend it to handle *alg* terms in our type-term language.

Colmerauer's method uses five reduction rules as follows:

- Compaction. Eliminate all the equations containing variable x in the left hand side and in the right hand side only the variable x is present.
- Variable Elimination. If the variable x and the variable y are distinct variables, and the equation $x{=}y$ is in the system, and x has other occurrences in the system, then replace all these other occurrences of the variable x by the occurrences of the variable y.
- Variable Anteposition. If there is an equation where its left hand side is a term which is not a variable and the right hand side is a variable x, then swap the order of the terms. The left hand side becomes the variable x and the right hand side the other term.
- Confrontation. If in the system of equations there is $x = t_1$ and $x = t_2$, where the size of t_1 is $\leq$ the size of t_2, then replace the two equations by $x = t_1$ and $t_1 = t_2$.
- Splitting. If the equation is of the form $f(t_1, ..., t_n) = g(r_1, ..., r_n)$, then if $f{=}g$ replace it by $t_1 = r_1$, $t_2 = r_2$,..., $t_n = r_n$, or if $f \neq g$ fail.

4.4.3 Algebraic Data Types

In Reduceron template code it is possible to use constructors to build low-level representations of high-level data structures like lists or trees for instance. The atoms CON and TAB in particular, encode computation involving algebraic data types. The work from Colmerauer only deals with function terms and term variables. In this section we explain how to solve an equation involving possibly recursive data types. First we give an informal approach; then we shall formalise it as a rule.

From the language of terms in Figure 4.7, we have that an algebraic data type is represented by the term *ics* where *ics* is a sequence of pairs representing indices and lists of argument types for constructors. In some cases we want to allow some extensions for the data types, we denote this idea by $\oplus ext$. For example the type for the atom BOOL is an algebraic data type containing two constructors. This type *cannot be extended*.

[(0,[]) , (1,[])]

```
        ----------------->   [(0,[]) , (1,[])]
        |
case sub  of
 alt0  ->  True    ------> [(1,[])] (+) (v1,[1]) --
                                                  |
                                                  =?
                                                  |
 alt1  ->  False   ------> [(0,[])] (+) (v2,[0]) --
```

FIGURE 4.8: Case example.

Now suppose we have an alternative in a case expression, the result type of this alternative should not be restricted. For example the type of a constant False compiled to an atom CON 0 0, is encoded as:

$[(0,[])] \oplus (v_e,[0])$

This term represents an algebraic data type which includes at least a 0-arity constructor with index 0. The possibility of other constructors is represented by the variable extension *ve*. Any algebraic data type assigned to *ve* must lack any constructor with index zero.

Let's outline how this idea works form in Figure 4.8. The case subject sub is assigned a *closed* algebraic data type, in this example it includes two constructors for alternatives alt0 and alt1. However, the types for True and False must be *open* allowing the possibility of an extension, so that there can be a solution to an equation requiring these two types be equal. The solution is to assign the type of False to the variable extension v_1, this is : $v_1 = [(0,[])] \oplus (v_e,[1,0])$, a new variable extension is added denoted by v_e. The type can be extended by any type *lacking* constructors [1,0]. The solution also assigns the type including True to the extension variable v_2 of False as follows: $v_2 = [(1,[])] \oplus (v_e,[1,0])$ and the explanation is similar to the previous one. A minor but important note is that extension variables v_e must be fresh variables and the same variable occurs in the type for alternative.

4.4.4 Rules for Solving Algebraic Data Types Equations

Now it is time to formalise these previous ideas for solving equations between algebraic data types. Figure 4.9 gives a reduction rule for equations between

two algebraic data types. We combine the algebraic type-form with and without extensions by introducing an empty extension. Here we use the convention $icsL$ to denote the index-constructor pairs ics in the left data type. Similarly, we use the same convention for the data type in the right $icsR$.

The exL and exR are the variable extension-lack information pairs. If we do not need to "extend" our data type, the extend information is not given. If we can extend the data type, then the extension information is a pair of a variable extension and the set of the lack information which contains the indexes of the constructors. We can extract the lack information using the auxiliary function $getlks$ to extract the second component of the extension.

We need first, to detect which ics exists in one data type and not in the other. The set difficsL are the existing ics in $icsL$ and not in $icsR$. For the set difficsR we apply the same principle but the other way around.

In general, the rule has two possibilities, one is a failure and the other possibility is to reduce the original equation to a set of derived equations.

The rule fails under any of the following conditions :

1. The types both have a constructor with the same index i but different arities.
2. There is a non-empty set difficsL (or difficsR), but the right (or left) data type *is closed*.
3. There is an index in the iL (or iR) that is already in the *lacks* set lsR (or lsL).
4. Both difficsL and difficsR sets are empty, and both types *are open* but their lacks sets are not equal.

Otherwise the rule succeeds. Derived equations are obtained as follows:

1. For each ics appearing in the left and also appearing in the right we construct equations between corresponding argument types.
2. We take into account the remaining constructors not found in that intersection as follows:

(a) If difficsL and difficsR are both non-empty we create two equations. The first assigns to the extension variable exR of the right a data type derived from the left. The new data type contains only the left constructors difficsL . The new extension information has a fresh variable, and the union of the two lacks sets. The second equation is similarly obtained, but with left and right exchanged.

(b) If difficsL is not empty but difficsR is empty we extend the data type of the right by assigning to its extension variable the data type derived from the left. It is a new data type containing the constructors in difficsL , with extension information *the same as* the data type in the left. This step only creates one equation.

(c) Similarly if difficsL is empty but difficsR is non-empty.

(d) The fourth possibility is when difficsL and difficsR are both empty. There are four sub-cases based on the extension information:

i. If both types are closed then we do not need to add any extra equation.

ii. If the type on the left is closed but the one in the right is open, a new equation assigns to the variable extension on the right an *empty* data type, with no constructors and no extension.

iii. Similarly, if the data type on the left is open and the one on the right is closed.

iv. Finally, if both data types are open, we create an equation between the extension variable on the left and the extension variable on the right.

4.4.5 Collecting Equations for `length` Function: An Informal Approach

This section uses a small example, a function to calculate the length of a given list, to explain informally the process of collecting equations for a given template. In Figure 4.10 we describe several stages of this process applied to the template code for the `length` function.

Before we proceed to type-check the template code, the code is checked by AtomCheck and PrimCheck analysis. The result is a split template, containing the

Given equation:

$$alg \quad icsL \oplus exL \quad = \quad alg \quad icsR \oplus exR$$

Fail under the following condition:

$$\begin{array}{l}
\textbf{if } \exists (iL, csL) \in icsL, (iR, csR) \in icsR : iL = iR \wedge |csL| \neq |csR| \\
\vee \\
(\mathit{diffsL} \neq \emptyset \wedge (null\, exR \vee \exists idlackL \in lacksL : idlackL \in lacksR)) \\
\vee \\
(\mathit{diffsR} \neq \emptyset \wedge (null\, exL \vee \exists idlackR \in lacksR : idlackR \in lacksL)) \\
\vee \\
(\mathit{diffsL} = \emptyset \wedge \mathit{diffsR} = \emptyset \wedge \neg null\, exL \wedge \neg null\, exR \wedge lacksL \neq lacksR) \\
\quad \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{diffsL} & = & \{(iL, cL) \mid (iL, cL) \in icsL, iL \notin icsR\} \\
\mathit{diffsR} & = & \{(iR, cR) \mid (iR, cR) \in icsR, iR \notin icsL\} \\
(veL, lacksL) & = & exL \\
(veR, lacksR) & = & exR
\end{array}
\end{array}$$

Otherwise, succeed replacing the equation by the following new equations:

$$ss_j = ts_j \quad , \forall i, j : \ (i, ss) \in icsL, (i, ts) \in icsR, 1 \leq j \leq |ss|$$

$$\begin{array}{lll}
veR = alg\ \mathit{diffsL}\ (v, lacksL \cup lacksR) & , \textbf{if } \mathit{diffsL} \neq \emptyset \wedge \text{difficsR } \neq \emptyset \\
veL = alg\ \mathit{diffsR}\ (v, lacksR \cup lacksL) & , \textbf{ditto}
\end{array}$$

$$\begin{array}{lll}
veR = alg\ \mathit{diffsL}\ exL & , \textbf{if } \mathit{diffsL} \neq \emptyset \ \wedge \ \mathit{diffsR} = \emptyset \\
veL = alg\ \mathit{diffsR}\ exR & , \textbf{if } \mathit{diffsL} = \emptyset \ \wedge \ \mathit{diffsR} \neq \emptyset
\end{array}$$

$$\begin{array}{lll}
alg\,[\,] = veL & , \textbf{if } \mathit{diffsL} = \emptyset \wedge \mathit{diffsR} = \emptyset \wedge \ \neg null\, exL \wedge \ null\ exR \\
veR = alg\,[\,] & , \textbf{if } \mathit{diffsL} = \emptyset \wedge \mathit{diffsR} = \emptyset \wedge \ \ null\, exL \wedge \ \neg null\, exR \\
veL = veR & , \textbf{if } \mathit{diffsL} = \emptyset \wedge \mathit{diffsR} = \emptyset \wedge \ \neg null\, exL \wedge \ \neg null\, exR
\end{array}$$

FIGURE 4.9: Rule to solve equations between two algebraic data types.

templates and the isolated integer blocks. A dependency analysis is performed to obtain the connected components in dependency order. Even if this appears to be a small example, it contains some interesting structures to deal with. For example, it contains a TAB i j k atom which represents references to other templates. In addition, we have a recursive call for the function length denoted by FUN 1 1 in the second template.

The first step is to calculate and give *skeleton types* to each template, see Figure 4.11. These skeleton types are just chains of term variables based on the arity of

The length code definition in Flite :

```
length (Cons x xs) = 1 + length xs
length Nil         = 0
```

The original template for length function :

```
(1,[ARG 0,TAB 1 2 0],[])
(3,[FUN 1 1,ARG 1,PTR 0],[[INT 1,PRI "(+)"]])
(1,[INT 0],[])
```

The split template for the length function after PrimCheck :

```
((1,[ARG 0,TAB 1 2 0],[]),[])
((3,[INT 0],[[]]),[[FUN 1 1,ARG 1]])
((1,[INT 0],[]),[])
```

The reordered split template after dependency analysis :

```
((3,[INT 0],[[]]),[[FUN 1 1,ARG 1]])
((1,[INT 0],[]),[])
((1,[ARG 0,TAB 1 2 0],[]),[])
```

FIGURE 4.10: Processing the *length* function's templates.

```
((3,[INT 0],[[]]),[[FUN 1 1,ARG 1]])
```

$a \rightarrow b \rightarrow c \rightarrow r_1$

```
((1,[INT 0],[]),[])
```

$d \rightarrow r_2$

```
((1,[ARG 0,TAB 1 2 0],[]),[])
```

$e \rightarrow r_3$

FIGURE 4.11: Skeleton types for the *length* function's split templates.

each template.

Once we have the skeleton types, we calculate the equations for the templates representing alternatives. In our example the alternatives are the first two templates (after reordering by dependency) as shown in Figure 4.12.

Recursive Case Template. For the template number one, which is a template of arity three, we assign the initial skeleton type as follows $a \to b \to c \to r_1$, a functional type based on the arity. We use the letter "r" subscripted as convention for the result type.

For the template one (3,[INT 0],[[]]), we only have as result type an Int. However, the type of the integer blocks denoted by [[FUN 1 1,ARG 1]] is an integer application of the function to the argument numbered one. We use the subscript "i" for type variables inside of integer blocks. In the equation $b_i = b \to d_i$ notice the b, which is the same "b" as in the skeleton type $a \to b \to c \to r_1$.

Note here we have a recursive call for length function in [FUN 1 1,ARG 1], the type for FUN 1 1 is the same type as the top-level definition of the template number one $e \to r_3$.

Type Invariance and Recursion. We use this invariant in our type- system: The recursive calls of a given function *must have* the same type as the type of the top level function definition of that function.

Base Case Template. The second template ((1,[INT 0],[]),[]) is easy to collect the equations from. The skeleton type for this template is a function $d \to r_2$. The r_2 is the result type and it is Int. There are no integer blocks at all, so there are no further equations from this template.

Main Template. Finally, and the most interesting part, there is the calculation of the equations for the main template for the length function itself. The skeleton type function for length is $e \to f$ where e is the type of the argument and f is the result type.

In Figure 4.13 first we detect that this template contains a *case table application.* This requires a different treatment from function applications. In the application example [ARG 0,TAB 2 2 0] everything before a TAB 2 2 0 atom is the case subject. In this example it is just a single argument ARG 0. In the TAB i j k

```
((3,[INT 0],[[]]),[[FUN 1 1,ARG 1]])
```

$a \rightarrow b \rightarrow c \rightarrow r_1$
$r_1 = Int$

$a_i = e \rightarrow r_3$
$a_i = b \rightarrow d_i$

```
((1,[INT 0],[]),[])
```

$d \rightarrow r_2$

$r_2 = Int$

FIGURE 4.12: Templates for the two alternatives of the *length* function, and their types.

```
   case subject part
          |
   _______|__________
  |                  |

((1,  [ARG 0             ,TAB 2 2 0],[]),[])

  |_____________________________|
                |
                |
          result type
```

$f = e \rightarrow r_3$

FIGURE 4.13: Template coding of the case expression in *length*.

constructor in this example the j component refers to the number of case alternatives, here there are two. The k component is zero, which tells us there are no auxiliary arguments in alternative templates (these would correspond to free variables in high-level case alternatives). The overall type is the result type of all the alternatives.

The type of the top level definition of length is defined by the equation $f{=}e \rightarrow r_3$. Then the equations for f are extracted in the following sequence:

- The type of ARG 0 is the type of e, which is the case subject type.

- Then, we construct this type as an algebraic data type, which contains two components (zero and one). The component one is just an empty component, from $d \rightarrow r_2$ we see that d is just a table. In the operational semantics §3.4 the table arguments are not used when a case alternative is reduced. The component zero contains the type variables a and b, which are taken from $a \rightarrow b \rightarrow c \rightarrow r_1$, where c is the table placeholder and r_1 is the result. Notice here the term variables in the components of case subject type are just instances of the types for the templates one and two as follows e = [(0,[a , b]),(1,[ ])]

- Now, for each alternative we extract the result. In our example the result types are Int and r_2. We assign those types to two fresh variables to construct two equations as follows:

  $$\begin{aligned} g &= Int \\ h &= r_2 \end{aligned}$$

  We extract all the type equations in the result type of each alternative. These equations are previously formed when we extract the types for the applications, for each application we generate two equations.

  $$r_1 = Int$$

  $$\begin{aligned} a_i &= e \rightarrow f \\ a_i &= b \rightarrow d_i \end{aligned}$$

- Finally, the result types from each alternative must be the same. We construct an equation $g = h$ to make the two result types equal. Additionally, we need to create an equation to say that the overall result type r_3 is equal to the result type of the alternatives, it is denoted by $f = g$ where g is the result type of one of the alternatives.

In the Figure 4.14 the final system of equations is given. The type variable t_0 represents the type of template zero, which encodes the function *length*. When we solve this system of equations in Figure 4.14 we obtain a simplified system as follows:

$$\begin{aligned} t_0 &= e \rightarrow f \\ e &= [(0, [a, e]), (1, [\,])] \\ f &= Int \end{aligned}$$

$$
\begin{array}{lcl}
t_0 & = & e \to f \\
e & = & [(0,[a,b]),(1,[\,])]
\end{array}
$$

$$
\begin{array}{lcl}
g & = & Int \\
h & = & r_2
\end{array}
$$

$$
r_1 = Int
$$

$$
\begin{array}{lcl}
a_i & = & e \to f \\
a_i & = & b \to d_i
\end{array}
$$

$$
\begin{array}{lcl}
g & = & h \\
f & = & g
\end{array}
$$

FIGURE 4.14: The final type equations extracted from template code for *length*.

Note the recursive structure of a list in the second equation. An algebraic data type with two constructors indexed by zero and one. The constructor zero has two arguments, the type variable a can be any type, and the variable e which is the recursive term at the algebraic data type. The constructor takes no arguments. The solution assigns to e the type of the *length* template argument.

We have seen in an informal way how to collect the equations for the *length* function,as an example. In addition we have shown the solution for that system of equations.

4.4.6 Rules for Collecting Equations

Here, we present the formal rules for collecting type equations from template code. The general outcome of the collection rules in Figures 4.15 and 4.16 is a pair (T, S), where T is the term-type which is the result type, S is the list of term-type equations related to T. The system S can be empty. For instance, the collected type information for an INT atom is : $(Int, [\,])$, where Int is the result type, and $[\,]$ the empty system of equations. The name of the collection rules has the prefix convention *TES* meaning type-term equations system.

In order to keep track of the types for the templates, we use an environment Γ, which is a list of pairs $(i, tsys)$, where i is the index of a template, and $tsys$ an associated type-term and equation system (T, S).

The steps to collect the equations are:

- Initialise the environment with a *skeleton type* for each template.
- For each template derive type-term equations from the integer blocks.
- For each template derive type-term equations from the spine application and the off-spine applications.
- Accumulate the equations.

We define the rule $TESSTemp$ to collect the type equations for a template, the rule $TESApply$ to collect equations for applications, $TESAtom$ to collect equations for each atom. The rule $TESApp$ *accumulates* the results of the application rule. The $TESCaseSubOrAtom$ rule checks if the atom examined is a TAB atom. If it is, then we extract the information before and after the TAB constructor, otherwise the atom is part of a function application.

4.4.7 Accumulating Applications

The simplest rule is the $TESApp$ which accumulates the results from the application rule $TESApply$, this rule traverses recursively all the structure of the application, reflecting the left-associative structure of a multi-argument application.

4.4.8 Collection of Equations in a Template

The $TESSTemp$ rule receives a term t, system of equations s, and the environment e. From now on we will use the variable name tes or the decorated version tes' to refer to such a triple. Additionally the rule takes the recursive connected components rgs and a split template.

The output of this rule has two possible alternatives:

1. If the template has only an empty spine application and an empty off-spine applications, the rule simply yields a fresh type variable, and the empty system of equations. This case can occur when there is an *undefined* template.

For instance, when the Flite compiler generates code for the *head* function, it creates automatically an alternative to handle the case when the list is empty. From the point of view of type-checking any *undefined* values must be compatible with any other type.

2. The second case is when we have a template, with a non-empty spine application. Apart from the standard application in template code we have a special case, when there is a TAB i j k in the list of atoms. If that occurs then anything before the tab constructor is considered as the case subject. In $TESSTemp$ the rule $TESCaseSubOrAtom$ is in charge to deal with such situation. This rule returns a tes' and the list of atoms in app'. Then it accumulates the applications in app' by using $TESApp$. By using this process, we eliminate the need to assign a type to a TAB as a first-class citizen atom.

4.4.9 The Application Rule.

The rule for application $TESApply$ introduces two equations $v_1 = t_1$ and $v_1 = t_2 \to v_2$, and the union of the derived system of equations s_1 and s_2 (See §4.4.6). The type-term variable v_2 contains the result type.

The atomic rules TESAtom are :

- for ARG i extracts the type of i from the environment.
- for FUN n i uses $TESFun$ to extract the type i from the environment. Depending on the context the extraction can yield an instance or a copy from the environment.
- for INT n, assigns as result type the Int type, and an empty set of type equations.
- for BOOL extracts a closed algebraic data type with two constructors, and an empty set of type equations.
- for CON j k, uses the rule $TESCon$ to extract a functional type based on the arity j, whose result type is an open algebraic type pairing k with the sequence of term variables that occurs in the functional type.
- for PTR i the rule for pointers $TESPtr$, extracts the type and the system of equations of the off-spine application at index i.

The rule for a case subject $TESCaseSub$ contains the construction of the alternative types from the values of the case alternative templates. The free-variables arguments following the TAB atom are equated with free variable types of each case alternative. The case subject is calculated with the information of the case alternatives. We equate the case subject calculated in $TESCaseSub$ rule *casesub* with the case subject *casesub* calculated in the rule $TESCaseSubOrAtom$. The result type of the case structure must be equal to the result types in the alternatives, in our case we select the first variable result vr_0.

4.4.10 Type Equations for Integer Blocks.

The int-block rule evaluates each application in an *integer block* and assign an Int type to those singleton applications containing just one argument. In that way, any integer arguments in int-blocks now have an explicit integer type.

4.5 Measuring the Effectiveness of TyreCheck

In Chapter 3 we described the machinery to measure the effectiveness of our static checkers. Here we explore the results of comparing TyreCheck against RunCheck. The overall process of testing is as follows :

- From an original well-behaved program in template code produce *n mutations*, then extract *randomly m mutations*.
- Keep only the *well-formed* mutations that are passed by AtomCheck.
- The filtered well-formed mutations are analysed by *PrimCheck*.
- If *PrimCheck* succeeds then the mutation is analysed by *TyreCheck*, otherwise the mutation is stopped after PrimCheck analysis.
- If *TyreCheck* can find a solution for the equations collected for each function definition, the mutation is classified *well-typed*, otherwise it is considered *ill-typed*.
- Evaluate the *well-formed* mutations by *RunCheck* (operational semantics). See §3.5.

$$
\begin{array}{l}
\mathit{TESSTemp}\ [\![\ st\ ]\!] \Rightarrow \\
\quad (v, [\,]) \qquad\qquad\qquad\qquad , \textbf{if}\ ap = [\,] \wedge aps = [\,] \\
\quad \mathit{TESApp}\ tes'\ [\![\ app'\ ]\!] \quad , \textbf{otherwise} \\
\quad \textbf{where} \\
\quad (tes', app') \qquad\qquad\qquad = \mathit{TESCaseSubOrAtom}\ [\![\ st\ ]\!] \\
\quad ((a, ap, aps), ibs) \qquad\qquad = st \\
\quad v \text{ is fresh}
\end{array}
$$

$$
\begin{array}{l}
\mathit{TESCaseSubOrAtom}\ tes\ [\![\ ((ari, a_1 \cdots a_n, apps), ibs)\ ]\!] \\
\quad \textbf{if}\ \forall j : \neg \mathit{isTab}\ a_j \qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \\
\quad\quad (\mathit{TESAtom}\ [\![\ a_1\ ]\!], a_2 \cdots a_n) \\
\quad \textbf{if}\ \forall m : 1 \leq m < i, \neg \mathit{isTab}\ a_m \wedge a_i = \text{TAB j k i} \qquad\qquad \Rightarrow \\
\quad\quad (\mathit{TESCaseSub}\ [j \cdots (j + k - 1)]\ l\ (cst, fvts), a_{i+l+1} \cdots a_n) \\
\quad \textbf{where} \\
\quad cst \quad = \mathit{TESApp}\ ([\![\ \mathit{TESAtom}\ [\![\ a_1\ ]\!]\ ]\!] a_2 \cdots a_{i-1}) \\
\quad fvts\ = \mathit{TESAtom}\ [\![\ a_{i+1}\ ]\!], \cdots, \mathit{TESAtom}\ [\![\ a_{i-l}\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathit{TESCaseSub}\ tid_0, \cdots, tid_n\ l\ [\![\ cst, fvts\ ]\!] \Rightarrow (vr_0, eqs) \\
\quad \textbf{where} \\
\quad (v_{0_i} \rightarrow \ldots v_{n_{i-1_i}} \rightarrow fv_{0_i} \rightarrow \ldots fv_{(l-1)_i} \rightarrow v_{r_i}, s_i) \qquad = \Gamma\ tid_i \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , 0 \leq i \leq n \\
\quad casesub = alg \quad (0, [v0_0, ..., vn_0]) \quad , ..., \quad (k-1, [v0_{k-1}, ..., vn_{k-1}]) \\
\quad pairsRes \qquad\qquad\qquad\qquad = \{v_{r_{i-1}} = v_{r_i} \mid 1 \leq i \leq n\} \\
\quad pairFvs \qquad\qquad\qquad\qquad = \{fv_{i_{(j-1)}} = fv_{i_j} \mid 1 \leq j \leq n \wedge 0 \leq i < l\} \\
\quad pairFvsAltTab \qquad\qquad\qquad = \{fv_{i_0} = fvsts_{0_i} \mid 0 \leq i < l\} \\
\quad (cst', css) \qquad\qquad\qquad\qquad = cst \\
\quad (fss_0 ... fss_n, fvsts_0 ... fvsts_n) \quad = fvts \\
\quad eqs \qquad\qquad\qquad\qquad\qquad = css \cup (cst', casesub) \cup s_0 \cup \cdots \cup s_n \cup pairRes \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup pairFvs \cup pairFvsAltTab \cup fss_0 ... fss_n
\end{array}
$$

FIGURE 4.15: Rules to extract equations from Template code.

- At the end there are six sets in our testing as described in Chapter 3:
  - The set `OK` is the set of *well-behaved* programs in *RunCheck*
  - The set `SC` is the set of *well-typed* programs satisfying *AtomCheck, PrimCheck and TyreCheck.*
  - The set `BP` is the set of bad programs allowed by static analysis but stopped during the execution by the run-time system.
  - The set `GP` is the set of *well-behaved* programs but stopped by the static analysis techniques.
  - The set `NT` is the set of (apparently) *non-terminating* programs.

$$
\begin{array}{l}
TESApp\ tes\ [\![\ a_1 \cdots a_n\ ]\!] \Rightarrow \\
\quad TESApply\ [\![\ (\cdots (TESApply\ tes\ [\![\ a_1\ ]\!]) \cdots)\ ]\!] a_n
\end{array}
$$

$$
\begin{array}{lll}
TESApply\ (t_1, s_1)\ [\![\ a\ ]\!] & \Rightarrow (v_2, s_3) & \\
\quad \textbf{where} & & \\
\quad (t_2, s_2) & = TESAtom\ [\![\ a\ ]\!] & \\
\quad s_3 & = \{(v_1 = t_1), (v_1 = t_2 \rightarrow v_2)\} \cup s_1 \cup s_2 & \\
\quad v_1, v_2 \text{ are fresh} &
\end{array}
$$

$$
\begin{array}{lll}
TESAtom & [\![\ \text{ARG x}\ ]\!] & \Rightarrow \Gamma x \\
TESAtom & [\![\ \text{INT i}\ ]\!] & \Rightarrow (Int, [\,]) \\
TESAtom & [\![\ \text{BOOL}\ ]\!] & \Rightarrow ([(0, [\,]), (1, [\,])], [\,]) \\
TESAtom & [\![\ \text{CON i j}\ ]\!] & \Rightarrow TESCon\ j\ k \\
TESAtom & [\![\ \text{FUN j k}\ ]\!] & \Rightarrow TESFun\ j \\
TESAtom & [\![\ \text{PTR x}\ ]\!] & \Rightarrow TESPtr\ x
\end{array}
$$

$$
TESCon\ j\ k \Rightarrow (v_0 \rightarrow v_1 \rightarrow v_{j-1} \rightarrow (alg[(k, v_0, ..., v_{j-1})](v_{ext}, k)), [\,])
$$
$$
\quad \textbf{where}
$$
$$
\quad v_0 \cdots v_{j-1}, v_{ext} \text{ are fresh}
$$

$$
TESPtr\ ((ari, app, ap_1 \cdots ap_n), ibs)\ x \Rightarrow TESSTemp\ [\![\ ((ari, ap_x, ap_1 \cdots ap_n), ibs)\ ]\!]
$$

$$
\begin{array}{ll}
TESFun\ id \Rightarrow & \\
\quad \Gamma\ id & , \textbf{if}\ \Gamma id = \bot \\
\quad instance\ id\ \Gamma & , \textbf{otherwise}
\end{array}
$$

FIGURE 4.16: Rules to extract type-equation systems from Template code, given type environment Γ.

– The set `AC` is the set of *well-formed* programs checked by AtomCheck.

4.5.1 Mutations : Delete

In this section we measure how effective PrimCheck and TyreCheck are when we test them against mutations in which we only *delete* an arbitrary atom. In Table 4.3 the results for testing PrimCheck are shown. In previous §4.3.4 the set of good programs `GP` was close to zero, in this test we preserve that result. For this experiment the amount of mutations is `200`. By deleting atoms we can damage the code seriously. The number of well-behaved programs (`OK`) is small in comparison with the number of well-formed mutations (AtomCheck).

Table 4.3: Mutations (200-Delete-PrimCheck)

Program	AtomCheck	PrimCheck	OK	GP	BP	NT
permsort	122	121	6	0	115	6
queens	117	105	4	0	101	13
queens2	193	186	18	1	169	7
ordlist	128	128	27	0	101	2
mss	111	105	1	0	104	10
parts	117	96	9	0	87	10
braun	173	171	8	0	163	6
adjoxo	194	182	9	0	173	6
tautology	182	171	3	0	168	18
cichelli2	188	183	10	0	173	12

Table 4.4: Mutations (200-Delete-TyreCheck)

Program	AtomCheck	PrimCheck	TyreCheck	OK	GP	BP	NT
permsort	122	121	7	6	3	4	6
queens	117	105	7	4	0	3	13
queens2	193	186	37	18	2	21	7
ordlist	128	128	6	27	23	2	2
mss	111	105	11	1	0	10	10
parts	117	96	23	9	1	15	10
braun	173	171	5	8	5	2	6
adjoxo	194	182	11	9	6	8	6
tautology	182	171	0	3	3	0	18
cichelli2	188	183	7	10	4	1	12

It is convenient to see the behaviour of *PrimCheck* and *TyreCheck* together. In Table 4.4 we show the results from applying first *PrimCheck* and then *TyreCheck* to the same set of mutations used in Table 4.3. The main goal is to reduce the number of bad programs `BP` yet preserve as many as possible of the good programs `GP`.

In `mss` there is a reduction in set `BP` from `104` to `10`, in this case the set of `BP` is `[95,100,75,95,72,72,72,72,100,75]`. As we can see there are repeated mutations, then the actual set is `[95,100,75,72]`. Even if the numbers in the set are duplicated in `mss`, the gain in the *BP* set when PrimCheck + TyreCheck are applied together is in factor of 10 times, than applying PrimCheck solely.

Table 4.5: Mutations (200-Increment-PrimCheck)

Program	AtomCheck	PrimCheck	OK	GP	BP	NT
permsort	137	133	40	0	93	2
queens	158	154	54	0	100	3
queens2	155	155	67	0	88	5
ordlist	149	146	70	2	78	2
mss	149	149	44	0	105	5
parts	131	131	48	0	83	8
braun	139	139	45	0	94	7
adjoxo	164	164	78	0	86	2
tautology	150	149	41	0	108	2
cichelli2	152	148	44	0	104	3

Table 4.6: Mutations (200-Increment-TyreCheck)

Program	AtomCheck	PrimCheck	TyreCheck	OK	GP	BP	NT
permsort	137	133	30	40	15	5	2
queens	158	154	51	54	6	3	3
queens2	155	155	73	67	11	17	5
ordlist	149	146	62	70	8	0	2
mss	149	149	51	44	1	8	5
parts	131	131	52	48	5	9	8
braun	139	139	40	45	9	4	7
adjoxo	164	164	64	78	23	9	2
tautology	150	149	41	41	6	6	2
cichelli2	152	148	45	44	5	6	3

4.5.2 Mutations : Increment

Tables 4.5 and 4.6 gives the results when the mutations are increments in the numeric values of the arguments of the atom constructors. Notice here how the set `OK` is bigger than in the mutations where we delete atoms. If we see the average of `BP` in 4.6, it is 7 bad programs allowed by PrimCheck + TyreCheck, and the average of well-typed programs is 51 (in column TyreCheck), that means that the static checkers are allowing around 1/7 of bad programs from the total of well-typed programs. And the average of `GP` stopped is almost 9, which means that we are stopping less than 1/6 of all the well-typed programs.

Table 4.7: Mutations (200-All-PrimCheck)

Program	AtomCheck	PrimCheck	OK	GP	BP	NT
permsort	169	164	39	0	125	2
queens	162	148	49	0	99	7
queens2	161	156	44	3	115	6
ordlist	166	163	47	1	117	2
mss	160	156	32	1	125	5
parts	149	144	55	2	91	12
braun	162	160	34	2	128	7
adjoxo	172	166	66	0	100	3
tautology	170	166	40	2	128	5
cichelli2	166	161	45	0	116	7

Table 4.8: Mutations (200-All-TyreCheck)

Program	AtomCheck	PrimCheck	TyreCheck	OK	GP	BP	NT
permsort	169	164	32	39	14	7	2
queens	162	148	43	49	8	2	7
queens2	161	156	45	44	10	11	6
ordlist	166	163	36	47	12	1	2
mss	160	156	34	32	5	7	5
parts	149	144	52	55	16	13	12
braun	162	160	31	34	8	5	7
adjoxo	172	166	58	66	19	11	3
tautology	170	166	34	40	9	3	5
cichelli2	166	161	44	45	8	7	7

4.5.3 Mutations : All

The results when we apply our static methods to a set of mutations in which we combine Deletion, Swapping and Increment `All` are in Tables 4.7 and 4.8. The damage produced by this kind of mutations is something in between deletion and increment. The average of well-typed programs is 41, and the average of `BP` is 7, that means that we are allowing 1/6 of bad programs.

4.5.4 Bad Guys and Good Guys

Here we discuss some examples of BP (Bad Programs) and GP (Good Programs). First, let us see the reasons for why some ill-behaved programs are allowed by TyreCheck. Let's see in detail the problems in the program `tautology`. Consider some mutations `[242,564,164,350]` in Figure 4.17 all of which belong to `BP`. The mutations presented here, are of the style `ALL`, which includes `Swap + Delete + Increment`.

For mutation 242, the function FUN 2 40 is the second alternative of a case table. The arity of the alternative in this example is the same as the arity of the case table function. Moreover, the result type of that alternative is a type-term variable, which can unify with any other type.

In the mutation 564, we have a similar problem, the function FUN 1 31 is again a branch of a case table, and has the same arity as the case table function, represented by FUN 1 29. And the result type, in this example is compatible with the result type of the complete case table function.

The mutation 164, is a similar case, except that here the TAB 11 2 2 makes reference to one branch of the complete case table function.

The mutation 350, is obvious, from the point of view of type-checking the logical primitives `(==)` and `(/=)` have the same type.

In summary the first three mutations in this example, the main problem is when a function (via TAB or FUN) jumps to a case alternative, and that case alternative is valid in terms of type-checking.

4.5.5 Tangled Functional Types

There are some term-type-graph tangled "solutions" that are perfectly valid, for instance $t = t \to t$. However, from the point of view of type-checking those tangled solutions yield bad programs allowed by the solver. One possible way to stop bad programs, is to detect when these tangled solutions are present, allowing only recursion in algebraic data types.

```
 (3,[FUN 2 38,PTR 0,PTR 1],[..]),                              original

 (3,[FUN 2 40,PTR 0,PTR 1],[..]))                              mutation 242


((1,[FUN 2 41,PTR 2,PTR 4],[[FUN 1 19,ARG 0],                  original
                            [FUN 1 29,PTR 0],
                            ...]),

 (1,[FUN 2 41,PTR 2,PTR 4],[[FUN 1 19,ARG 0],
                            [FUN 1 31,PTR 0],                  mutation 564
                            ...]))


 ((4,[FUN 2 7,ARG 3,ARG 0,TAB 8 2 2,ARG 3,ARG 1],[]),          original

  (4,[FUN 2 7,ARG 3,ARG 0,TAB 11 2 2,ARG 3,ARG 1],[]))         mutation 164


 ((1,[INT 0,PTR 0,TAB 26 2 1,ARG 0],[[ARG 0,PRI "(==)"]]),     original
  (1,[INT 0,PTR 0,TAB 26 2 1,ARG 0],[[ARG 0,PRI "(/=)"]]))     mutation 350
```

FIGURE 4.17: Well-typed but ill-behaved template code in `tautology` mutants.

```
 (0,[CON 2 0,INT 97,PTR 0],[[CON 2 0,INT 98,CON 0 1]]),        original

 (0,[CON 2 0,INT 97,PTR 0],[[INT 98,CON 2 0,CON 0 1]])         mutation 1160


 (0,[CON 2 0,INT 97,PTR 0],[[CON 2 0,INT 98,CON 0 1]]),        original

 (0,[INT 97,CON 2 0,PTR 0],[[CON 2 0,INT 98,CON 0 1]]))        mutation 1140


 (3,[CON 2 0,ARG 0,PTR 2],[[FUN 2 28,ARG 0],
                           [FUN 2 49,PTR 0,ARG 1]...]),        original

 (3,[CON 2 0,ARG 0,PTR 2],[[FUN 2 28,ARG 0],
                           [PTR 0,FUN 2 49,ARG 1]...]))        mutation 523


((3,[FUN 2 1,ARG 0,ARG 2],[]),                                 original

 (3,[ARG 0,FUN 2 1,ARG 2],[]))                                 mutation 224
```

FIGURE 4.18: Some well-behaved but ill-typed template code in `tautology` mutants.

4.6 Summary

The main contributions of this Chapter are:

- An equation based type-checker for low-level code capable of dealing with recursive structures without explicit type information.
- A method to eliminate the need for dependent types called PrimCheck.

We have shown that it is possible to apply static analysis techniques to Reduceron code. Even in the absence of *extra annotations* the results of applying *PrimCheck* combined with *TyreCheck* can stop ill-behaved programs.

In addition, we have shown how successful the static techniques are in detecting ill-behaved programs when we present different kinds of mutation scenarios.

In the subsequent chapters we will see the compatibility between low-level type-checker and high-level type-checker. In addition, we will see how to make the static-checker tools faster and less expensive in terms of memory usage.

Chapter 5

Type Compatibility

5.1 Introduction

In this Chapter we explore the compatibility between high-level and low-level types. The high-level type information is inferred by a type-checker for Flite. The low-level type information is derived from the type-terms equations produced by Reduceron TyreCheck.

The road map of this Chapter is as follows: In §5.2 we depict our overall idea for a principle of compatibility, in §5.3 we give the language for high-level types, then in §5.4 we describe the language of the low-level types. In §5.5 we give the details to translate high-level types to low-level types. In §5.6 we give some examples of how we translate from high-level types to low-level term-types. Finally, in §5.7 we give a discussion and in §5.8 we present some results from the experiments.

5.2 Principles of Compatibility

In Figure 5.1 the source code is represented by the bullet circle in the upper-left corner. At program level, we can *Compile* source Flite code. The transformation schemas for the compiler from Flite to Reduceron code are explained in §2.7 of paper [72].

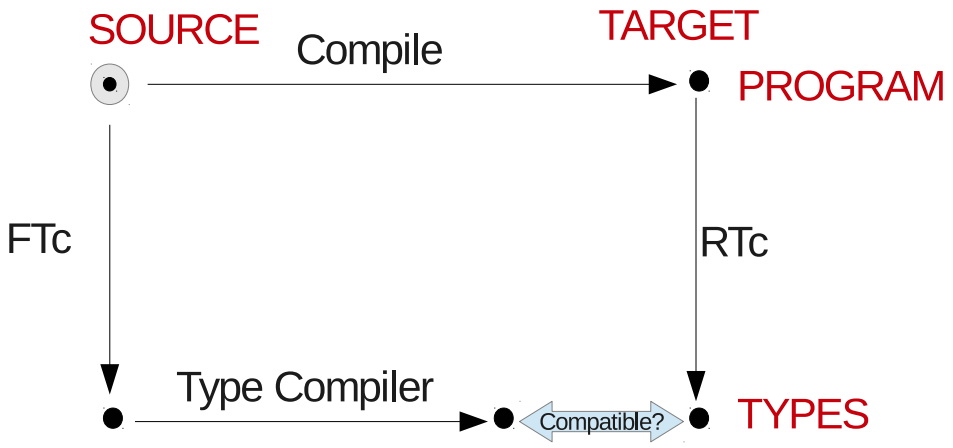


FIGURE 5.1: Type compatibility diagram.

An Flite type checker infers type information from the context of the function definitions, and explicitly given data type information. It uses standard type constructors and it is mostly based on Chapters 8 and 9 of [2], extended to handle case constructors, and algebraic data types. In the diagram the type-checker for Flite is denoted by *FTc*. More details of *FTc* are given in §5.3.

On the right-hand side of the diagram we have the type inference system for Reduceron code *RTc*. It is a type-term equations based system. The type information is extracted from Reduceron template code. In the presence of recursive calls in algebraic data types (ADT), the derived equations between type-terms are also recursive. More of the details are given in §5.4.

Although the two type representations are different in the high-level and low-level type systems, we wish to demonstrate a compatibility between them. We therefore translate the high-level types into a low-level representation. In the diagram we call the translator a *Type Compiler*.

In order to test whether high-level and low-level types are compatible we supply to the TyreCheck *solver* a combined system of equations including the systems of equations from the two systems. If this combined system of equations can be solved, the two systems are compatible. In the diagram we denote this test by the two ways arrow *Compatible?*. The results and discussions of the compatibility test are given in §5.7.

$$
\begin{array}{rcl}
ht :: & = & hv \quad | \quad C \quad id \quad ht \\
h\Gamma :: & = & \overrightarrow{id \quad \times \quad ht} \\
 & & \\
hdt :: & = & id \quad \times \quad \overrightarrow{n} \quad \times \quad \overrightarrow{Ct} \\
Ct :: & = & id \quad \times \quad \overrightarrow{ht} \\
hdts :: & = & \overrightarrow{hdt}
\end{array}
$$

FIGURE 5.2: High-level type language.

5.3 High Level Types

In Figure 5.2 the type language for Flite includes three main components : types, type environment and data type definitions. A high-level type ht is either a type variable hv, or a type constructor C with name id, and a list of types ht which may be empty.

A type environment $h\Gamma$ is a list of pairs, each pair contains the function definition name denoted by id, and its corresponding type ht.

Finally, we have dts which is a list of data type definitions dt. Each data type dt definition, is a triple containing a name i, a list of arguments denoted by $\vec{n}$, and a list of constructors Ct. Each constructor Ct is a pair of id and a list of types ht. The ids are sorted alphabetically in each dt to preserve the order of the indexes of the constructors. This order is used by the compiler from Flite to Reduceron code, so it is essential for compatibility with the sequence of indexes in low-level types.

The type inference system for Flite provides a type environment and the list of data types if the program is well-typed.

5.3.1 Examples of Types and Data Type Definitions

A concrete representation for integers is encoded as `C "Int" []`, a function type Int applied to zero arguments. A function type is encoded as `C "->" [hv1,hv2]`. A function type $Int \rightarrow Int$ is encoded as `C "->" [C "Int" [],C "Int" []]`.

The encoding of the data type declarations in source language deserves more explanation. Suppose we have two data type definitions for `List` and `Weights` as follows:

- `data List a = Cons a (List a) | Nil`.
- `data Weights a b = Weights a b`.

The encoding for List are :

id = "List", $\vec{n}$ = [a], Ct_0 = ("Cons" [a,List a]) and Ct_1 = ("Nil",[ ]). The type for List has the recursion in the second argument of Cons denoted by (List a).

Similarly for Weights we have : id= "Weights", $\vec{n}$= [a,b], and Ct_0 = ("Weights",[a,b]).

5.4 Low-Level Types

In Figure 5.3 the low-level type language contains type-terms denoted by lt. The type-terms are either a type variable lv, or an integer Int, or a function type $lt \rightarrow lt$, or an *open* algebraic data type denoted by $alg\ \overrightarrow{ics}\ ext$, or a *closed* algebraic data type denoted by $alg\ \overrightarrow{ics}$. The $\overrightarrow{ics}$ is a list of pairs associating with each index i of a list of that constructor's argument types $\vec{lt}$. Recalling from Chapter 4, algebraic data types in some cases need to be open to allow any further constructor components or closed to restrict them. The extension ext in Figure 5.3 is the information that tells which are the lacks indexes of the alternatives of that data type. If the data type is closed then no ext information is provided.

Then we define eqn, which contains two terms, the left and the right hand sides of an equation. The definition sys denotes a list of equations $\overrightarrow{eqn}$. The term-sys pair is denoted by $tsys$, it is a convenient way to represent the solutions. Finally a low-level type environment $\overrightarrow{id \times tsys}$ is a list of pairs, and each pair contains a template index and a term-sys pair $tsys$, denoting the computed type.

$$
\begin{array}{rcl}
lt :: & = & lv \mid Int \mid lt \rightarrow lt \mid alg\ \overrightarrow{ics} \oplus ext \mid alg\ \overrightarrow{ics} \\
ics :: & = & i \times \vec{lt} \\
ext :: & = & lv \times \vec{i} \\
\\
eqn :: & = & lt \times lt \\
sys :: & = & \overrightarrow{eqn} \\
tsys :: & = & lt \times sys \\
\\
l\Gamma :: & = & \overrightarrow{id \times tsys}
\end{array}
$$

FIGURE 5.3: Low-level type language.

5.5 From High-level to Low-level Types

Now that we know the language for the high-level and low-level types, the first step towards compatibility is to compile Flite types to low-level types for each function definition involved in the source code.

When we compile the Flite source code to Reduceron template code, we have three pieces of information:

1. The data type declarations.
2. The inferred types for each function definition (as given by a high-level type-checker).
3. The actual template code.

Compilation of high-level types to low-level type-terms and equation systems consists of two phases:

- The compilation of the data type declarations.
- The compilation of the inferred types in each function definition.

5.5.1 The Compilation of Data Type Declarations

In Figure 5.4 the rule CE compiles the high-level environment $l\Gamma$ into a low-level environment $h\Gamma$. It takes all the data type definitions denoted by $hdt_0 \ldots hdt_n$ and gives back the low-level environment $< id \times tsys > \cup CES$. Recall that our low-level environment is a list of pairs where each pair is a function identifier and a term-sys pair. Each term-sys pair $tsys$ is compiled by using the data type compilation rule DT. The rule CE is applied recursively to each data type definition, that is denoted in the definition of CES.

The rule DT translates each high-level algebraic data type definition into a low-level encoding of the type. It takes as auxiliary argument the current environment env, the list of all data type definitions ds, and the term variable v for the left hand side of the first equation. Knowing which term variable is assigned to each algebraic data type helps us when recursive references occur in constructor type $\overrightarrow{ct}$.

The TS rule in Figure 5.4 translates the terms found in each constructor inside a data type declaration. The rule TS applies the translation rule T in Figure 5.5 to translate each term in the declarations.

Now in Figure 5.5 we have the rule T. If it is applied to a high-level term variable v or the Int type it simply returns a pair of that type with an empty list of equations. Now the interesting part is the translation of constructors. In this rule, we need to take into account that there are recursive calls in the terms inside of the constructors. For that reason we use two variables v and v', we use v' when we find recursion in the list of terms in each constructor. To keep our rules simple, we update the low-level environment when we make a translation. That is why we use the data type transformation rule DT if a given definition is not in the environment or the auxiliary function $fromEnv$ which takes a copy of the data type constructor found in the environment.

In the rule T we don't have functions as types. Flite source does not have the ability to use function types in the data type declarations.

$$
\begin{array}{lll}
CE < h\Gamma, v > [\![\ hdt_0 ... hdt_n\ ]\!] & \Rightarrow < id \quad , \quad tsys > \cup CES & \\
\quad \textbf{where} & & \\
\quad tsys & = DT < h\Gamma, hdt_0 ... hdt_n, v > [\![\ hdt_0\ ]\!] & \\
\quad CES & = \overrightarrow{CE} < h\Gamma, v > [\![\ hdt_1 ... hdt_n\ ]\!] & \\
\quad (id, \vec{n}, \overrightarrow{ct}) & = \overrightarrow{hdt_0} &
\end{array}
$$

$$
\begin{array}{lll}
DT < h\Gamma, hdts, v > [\![\ (id, \vec{n}, \overrightarrow{ct})\ ]\!] & \Rightarrow < v, (v = alg\ ics) \cup sys > & \\
\quad \textbf{where} & & \\
\quad (id_0, \overrightarrow{ht_0}) ... (id_n, \overrightarrow{ht_n}) & = \overrightarrow{ct} & \\
\quad (t_{00}, sys_{00}) ... (t_{0n}, sys_{0n}) & = TS < h\Gamma, hdts, id, v > [\![\ \overrightarrow{ht_0}\ ]\!] & \\
\quad \vdots & & \\
\quad (t_{m0}, sys_{m0}) ... (t_{mn}, sys_{mn}) & = TS < h\Gamma, hdts, id, v > [\![\ \overrightarrow{ht_m}\ ]\!] & \\
\quad sys & = sys_{00} ... sys_{0n} ... sys_{m0} ... sys_{mn} & \\
\quad ics & = (0, t_{00} ... t_{0n}) ... (m, t_{m0} ... t_{mn}) &
\end{array}
$$

$$
\begin{array}{lll}
TS < h\Gamma, hdts, id, v > [\![\ \overrightarrow{t_0 ... t_n}\ ]\!] & \Rightarrow tsys_0 ... tsys_m & \\
\quad \textbf{where} & & \\
\quad tsys_0 & = T < h\Gamma, hdts, id, v, v_0 > [\![\ t_0\ ]\!] & \\
\quad \vdots & & \\
\quad tsys_m & = T < h\Gamma, hdts, id, v, v_m > [\![\ t_m\ ]\!] &
\end{array}
$$

FIGURE 5.4: Translation of data type declarations (I).

$$T < h\Gamma, ds, id, v, v' > [\![\ v\ ]\!] \Rightarrow < v, [\,] >$$

$$T < h\Gamma, ds, id, v, v' > [\![\ Int\ ]\!] \Rightarrow < Int, [\,] >$$

$$
\begin{array}{l}
T\ h\Gamma, ds, id, v, v' [\![\ C \quad id' \quad ts\ ]\!] \\
\Rightarrow \quad < v', [\,] > \quad \textbf{if} \quad id = id' \\
\Rightarrow \quad < v', s_0 ... s_n > \quad \textbf{if} \quad id \neq id' \wedge id' \in ds \wedge v' \in s_0 ... s_n \\
\Rightarrow \quad < v, s_0 ... s_n > \quad \textbf{if} \quad id \neq id' \wedge id' \in ds \wedge v' \notin s_0 ... s_n \\
\Rightarrow \quad < t_0, s \cup s_0 ... s_n > \quad \textbf{if} \quad id \neq id' \wedge id' \notin ds \\
\textbf{where} \quad (t, s) \\
= DT < h\Gamma, ds, hdts >, \textbf{if} id' \in h\Gamma \\
= fromEnv \quad h\Gamma \quad id' \\
(t_0 ... sys_0) ... (t_n ... sys_n) = TS < h\Gamma, ds, id, v, v' > ts
\end{array}
$$

FIGURE 5.5: Translation of data type declarations (II).

5.5.2 Compiling Types for Each Function Definition

The type information provided by the Flite type-checker has a list of function top-level definitions, and their corresponding types. In this section we will see how to compile such types.

To make use of the C and CS rules in Figure 5.6 for compiling term and a list of terms, we need first to translate the data type declarations as in §5.5.1 provided by the Flite type system. By using the low-level environment we can extract an instance of a given data type when we need it. The general outcome of the following rules is a $tsys$ tuple formed by a term and a system of term equations.

In Figure 5.6 the rule CS applies the rule C over a list of high-level types. The rule C, translates a high-level type expression to a pair of term and system of equations. The first compilation option corresponds to the type variables, when it occurs then the new term-sys pair is just a new term-type variable, and an empty system of equations, denoted by $< v, [\,] >$, where v is a fresh variable name. The translation for int is straight forward, it becomes a term-type with an empty set of equations $< int, [\,] >$. When we want to translate a function denoted by $t_1 \to t_2$, we compile recursively each type separately and extract two new term-types t_1 and t_2. From the compilation of t_1 and t_2 we extract also two sets of term equations sys_1 and sys_2. Finally we build a functional type and the concatenation of the two sets of term equations as follows $< t'_1 \to t'_2, sys_1 \cup sys_2 >$. The last part of the compile rule deals with the data type constructors. It *assumes* that in the current environment env the constructor denoted by id already exists. From the environment env the compiler takes an instance for the constructor name id. Then it computes the list of type expressions that are in the high-level constructors.

5.6 Examples of Translations.

5.6.1 An Example of Translation for Bool Data Type

Assume we a have a data type definition `Bool = False | True`. Here id=Bool, $\vec{n}$=[ ], and Ct_0 = (False,[ ]), Ct_1 = (True,[ ]) we can translate to a low-level algebraic data type as:

alg (0,[ ]) (1,[ ])

$$CS < h\Gamma, hdts, v > [\![\overrightarrow{ht}]\!] \Rightarrow tsys_0 ... tsys_m$$
where
$t_0 ... t_n = \overrightarrow{ht}$
$tsys_0 = C < h\Gamma, hdts, v_0 > [\![t_0]\!]$
$\vdots$
$tsys_m = C < h\Gamma, hdts, v_m > [\![t_m]\!]$

$$C < env, ds, v > [\![v]\!] \Rightarrow < v', [\,] >$$

$$C < env, ds, v > [\![Int]\!] \Rightarrow < Int, [\,] >$$

$$C < env, ds, v > [\![t1 \rightarrow t2]\!] \Rightarrow < t1' \rightarrow t2', sys1 \cup sys2 >$$
where
$< t1', sys1 >= C < env, ds, v' > [\![t1]\!]$
$< t2', sys2 >= C < env, ds, v'' > [\![t2]\!]$
$v', v'' \quad are \quad fresh \quad vars$

$$C < env, ds, v > [\![C \quad id \quad \vec{t}]\!] \Rightarrow$$
$< t1', sys1 \cup (t2_0 = t2_0) ... (t2_n = t2_n) \cup sys2_0 ... sys2_n >$
where
$< t1', sys1 >= instance < id, env >$
$\overrightarrow{< t2, sys2 >} = CS < env, ds, v''' > [\![\vec{t}]\!]$
$v''' \quad is \quad fresh \quad var$

FIGURE 5.6: Compilation rules for types.

In addition, notice in the definition Bool = False — True there are no arguments for the type Bool, which indicates that there are no polymorphic variables in this data type.

Each high-level data type definition has a name *id*, a list of polymorphic variables denoted by $\vec{n}$, and a list of pairs (index,constructors) denoted by $\vec{C}t$, each constructor can be of any type. In order to translate the high-level type, we also need a list of data type definitions denoted in *ds*, and the type environment denoted by *env*.

Notice that in the data type definitions we have a complete list of data type constructors, so we do not allow any type extension. The list of index constructors *ics* is extracted from the compilation of each single index constructor.

The two constructors are compiled to obtain *ics* [(0,[ ]),(1,[ ])], the index zero is the False constructor, and the one is True constructor. Because there are no components, the list of components for each constructor is empty.

5.6.2 Example of Type Compatibility of *map* Function

Our basic method to check if the types are compatible is in three steps:

- Compile high-level types of the map function to low-level term-sys pair.
- From the compiled term-sys (t_1, sys_1) and inferred term-sys (t_2, sys_2) equate t_1=t_2 and append the two sys parts to the system of equations. We will have a system of equations as follows: $t_1 = t_2 \cup sys_1 \cup sys_2$.
- Solve the system of the equations.

Let us consider the *map* function in high-level type representation in Figure 5.7, it is $(a \to b) \to [a] \to [b]$ a function that takes a function from $(a \to b)$ as the first argument, and a list of $[a]$ as its second argument. The result type is $[b]$. In the compiled types in Figure 5.7 we have type term and a list of equations. The type term $(a \to b) \to c \to d$ is the type for $(a \to b) \to [a] \to [b]$. The types $[a]$ and $[b]$ are represented by c and d respectively in low-level terms. Now if we look in the set of equations in the compiled type, we find out that c and d are the left hand sides of two equations. Those two equations represent the list structures of $[a]$ and $[b]$. In this example the high-level type $[a]$ is represented by the low-level type equation c =(0:[e , c]) + (1:[ ]). That is a recursive equation where the variable c is in both sides of the equation. And more precisely this recursion occurs in the component number zero, in its second term-type variable c. The constructor one contains no term-types constructors. This is the representation of the *nil case*. Finally, because we know which are the constructors allowed in this data type declaration, the lists cannot be extended. The translation of $[b]$ is similar, here $[b]$ is translated to the equation d=(0:[f ; d]) + (1:[ ]).

We can see the type of map function inferred by TyreCheck. The representation is quite close to the compiled version, except that we added an extra equation to represent a_1, that is the first equation in the list of the system of equations $d_1 = e_1 \to f_1$ and some other accumulated equations.

Once we have the compiled types for `map` function and the inferred types from Reduceron, we can follow with the second step, which is equating the two type-terms and appending the two systems of equations. Equate $(a \to b) \to c \to d = a_1 \to b_1 \to c_1$ and append the two systems of equations, see in Figure 5.7 the list of equations to be solved. Notice the first equation $(a \to b) \to c \to d = a_1 \to b_1 \to c_1$ which is the link to the remaining set of equations.

Once we have the system of equations we solve them to obtain a solution.

The solution for the system of equations includes two sets, the first set is for var terms assigned to any non-var term. The second set is a var term to var term assignments. Here, we make explicit the non-extension at the end of the data types. The extension is denoted by a variable and the list of lack indexes as in $v_2[1, 0]$ for instance.

5.7 Type Compatibility and Discussion

The main goal of this Chapter is to explore the compatibility between Flite types and Reduceron types. The first attempt to express this type compatibility is when the two combined systems of equations have at least one solution. This is, by applying the solver described in Chapter 3 we can decide if the systems of equations of two different universes are compatible. The idea of the compatibility is expressed in the diagram in Figure 5.8. From Flite source code we compile to Reduceron code red, we also infer the type information by using the type-checker for Flite. From the compilation and type-checking at high-level we obtain top-level function definitions f with the high-level types denoted by hltypes. Then we use compileTypes to translate the high-level types to low-level as (f,(t,sys)), where f is the *id* of the top-level function definition, and (t,sys) is the *tsys* pair to represent the type of the function f and all the possible equations related to that type.

TyreCheck computes the type information as (f',(t',sys')). Finally, the solve function computes a possible solution for the system of equations obtained from the

High-level type of map function:

$$(a \to b) \to [a] \to [b]$$

Compiled types:

$$\begin{array}{lll} ((a \to b) \to c \to d, & [& c = (0 : [e; c]) + (1 : [\,]) \\ & & e = g \\ & & d = (0 : [f; d]) + (1 : [\,]) \\ & & f = h]) \end{array}$$

Reduceron types:

$$\begin{array}{lll} (a_1 \to b_1 \to c_1, & [& d_1 = e_1 \to f_1 \\ & & b_1 = (0 : [e_1; b_1]) + (1 : [\,]) \\ & & g_1 = (0 : [f_1; h_1]) + v_2[1, 0] \\ & & h_1 = (1 : [\,]) + g_1[1] \\ & & c_1 = h_1 \\ & & a_1 = d_1]) \end{array}$$

Equations to be solved:

$$\begin{array}{rcl} (a \to b) \to c \to d & = & a_1 \to b_1 \to c_1 \\ c & = & (0 : [e; c]) + (1 : [\,]) \\ e & = & g \\ d & = & (0 : [f; d]) + (1 : [\,]) \\ f & = & h \\ d_1 & = & e_1 \to f_1 \\ b_1 & = & (0 : [e_1; b_1]) + (1 : [\,]) \\ g_1 & = & (0 : [f_1; h_1]) + v_2[1, 0] \\ h_1 & = & (1 : [\,]) + g_1[1]) \\ c_1 & = & h_1 \\ a_1 & = & d_1 \end{array}$$

Solution:

$$\begin{array}{rl} d_1 & = g \to f_1 \\ g_1 & = (0 : [f_1; d]) \\ c & = (0 : [g; c]) + (1 : [\,]) \\ d & = (1 : [\,]) + g_1[1]) \end{array}$$

$$\begin{array}{lllll} a = g & b = f_1 & e_1 = g & b_1 = c & h = f_1 \\ h_1 = d & c_1 = d & a_1 = d_1 & e = g & f = f_1 \end{array}$$

FIGURE 5.7: Example for type compatibility of the `map` function.

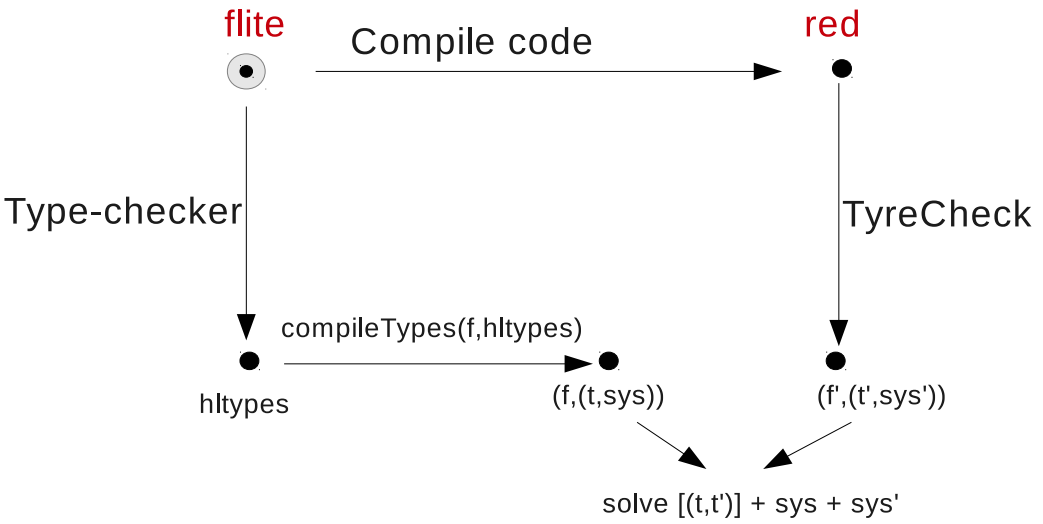


FIGURE 5.8: The actual model for type compatibility.

compilation of high-level types and the low-level types inferred by TyreCheck. In the diagram it is denoted by the function solve [(t,t')]+sys+sys'.

We compare the two sets of equations for each function definition type, and if there exists a solution for them, then the types are compatible. In this validation, we do not consider the implicit problem of extending or not the algebraic data types (eg. Sub-typing). The type compatibility between the low-level and high-level types is a condition to attest that the low-level types are *compatible* to the types in the high-level source language. There are some considerations that we must deal with, for example in the extension in the algebraic data types we allow the compatibility between more specific and more general types. In some sense when we talk about algebraic data types, the result type can be *more relaxed* and the argument types *less relaxed*. For instance, in a case expression we can restrict the number of case alternatives by assigning to them a closed algebraic data type.

5.8 Results

In Table 5.1 we have the results of testing two main properties when we solve the systems of equations in compile types and the inferred types from Reduceron. The

Program	Prop1	Prop2
PermSort	True	True
Queens	True	True
Queens2	True	True
MSS	True	True
OrdList	True	True
PermSort	True	True
Parts	True	True
Sudoku	True	True
Taut	True	True
Cichelli	True	True
While	True	True

TABLE 5.1: Property test over set of programs.

first property tell us if the system has a solution. The second property checks that once we have a solution, if we apply the substitutions of the solution to the original systems (the system from the compilation and the system from TyreCheck) we get the same term and list of term equations, for each top-level function definition.

5.8.1 Type Compatibility Results and Discussion

In Figure 5.9 we have the compiled types and the inferred types for the append function. We know the type for append is $[a] \to [a] \to [a]$ at source level. The types for the low-level encoding are represented by a (t, sys) pair, where t is a term and sys is a system of equations. Consider the structure of the compiled data type for the second argument of the append function:

$$b = (0 : [f; b])(1 : [\,])$$

The data type has two constructors, and the constructor type indexed by zero contains a recursive term b. For this particular case there is *less* information in the inferred type. The type for the second argument of append in the inferred type, k is $(0 : [m; n]) + (o, [0]))$ and by the context $k = n$ the recursive term. There is no constructor with index one.

In the solution set the equation $o = (1 : [\,]))$ tells us the inferred type is *compatible* with the compiled one, as it can be extended to include a nullary constructor with index one.

The compiled high-level type (t',s') is:

$$(a \to b \to c \quad ,\{ \quad \begin{array}{l} a = (0 : [d; a])(1 : [\,]) \\ d = e, \\ b = (0 : [f; b])(1 : [\,]), \\ f = g, \\ c = (0 : [h; c])(1 : [\,]), \\ h = i\}) \end{array}$$

The inferred low-level type (t,s) is:

$$(j \to k \to l \quad ,\{ \quad \begin{array}{l} j = (0 : [m; j])(1 : [\,]), \\ n = (0 : [m; n]) + (o, [0]), \\ l = n, \\ k = n\}) \end{array}$$

The solution demonstrating compatibility is:

$$\begin{array}{llllllllll} \{ & o & = & (1 : [\,]), & a & = (0 : [e; a])(1 : [\,]), & b & = (0 : [e; b])(1 : [\,]) \\ & g & = & e, & j & = a, & m & = e, \\ & c & = & b, & i & = e, & n & = b, \\ & l & = & b, & k & = b, & d & = e, \\ & f & = & e, & h & = e\} & & \end{array}$$

FIGURE 5.9: An example of a type compiled from high-level F-lite type and a type inferred from low-level code, for the append function, along with a solution showing these two types are compatible.

Often, as in this example, inferred types are more general than compiled ones. They specify only some constructions but they are open to extension.

As Figure 5.9 illustrates it is typically apparent that there is a similar structure in both systems of equations. We have similar data types and a similar number of equations. We point this out because we could have, in one system, a list of equations containing complex type terms and, in the other system, equations containing just distinct variables! Of course we can have a solution in that case. How can we have some confidence that we do not have this case? We can at least count the *number* of variables in both systems, and measure the *size* of the systems. In Table 5.2 we have a measure for variables, sizes and time to solve the systems of equations for all top-level function definitions in the benchmark programs.

TABLE 5.2: Measures of the type-equation systems obtained for all top-level definitions, both by inference from low-level code and by compilation from high-level types, along with the time needed for the Haskell model of TyreCheck to verify compatibility.

	Number of vars		Total size of type-terms		
	Inferred	Compiled	Inferred	Compiled	
Program					Time(s).
Queens	88	54	223	193	0.69
Queens2	196	143	451	499	2.52
MSS	86	67	219	219	0.96
PermSort	79	69	202	206	1.00
OrdList	73	51	173	151	1.64
While	225	158	712	1103	13.32
Sudoku	504	367	1202	1227	31.38
Parts	94	60	237	211	0.96
Braun	147	105	339	322	5.47
Taut	210	105	494	365	4.92
Cichelli	480	490	1164	1678	32.02

5.9 Summary

In this chapter we have shown the compatibility between high-level type information and low-level type information. The high-level types are inferred by the Flite type-checker. The high-level types were compiled to a low-level type representation in order to be compared with the inferred types by Reduceron TyreCheck.

We have shown the principle of compatibility between high-level and low-level types, which is given by finding a solution for the combined sets of a system of equations. We have established the basis for the relation of the low-level and high-level types. More properties of this correspondence can be explored and justified in a future work.

Chapter 6

A More Efficient Implementation

6.1 Introduction

In this Chapter we measure the *space* and *time* required to apply TyreCheck to Reduceron code. The speed of the checking is one of our main concerns. In §6.2 we discuss how space and time costs are significant to our work. Then, we explain the time and space costs of the Haskell prototype called *TyreCheckH* , in §6.3. In §6.4 we give an overview of how we translate the Haskell model to a C

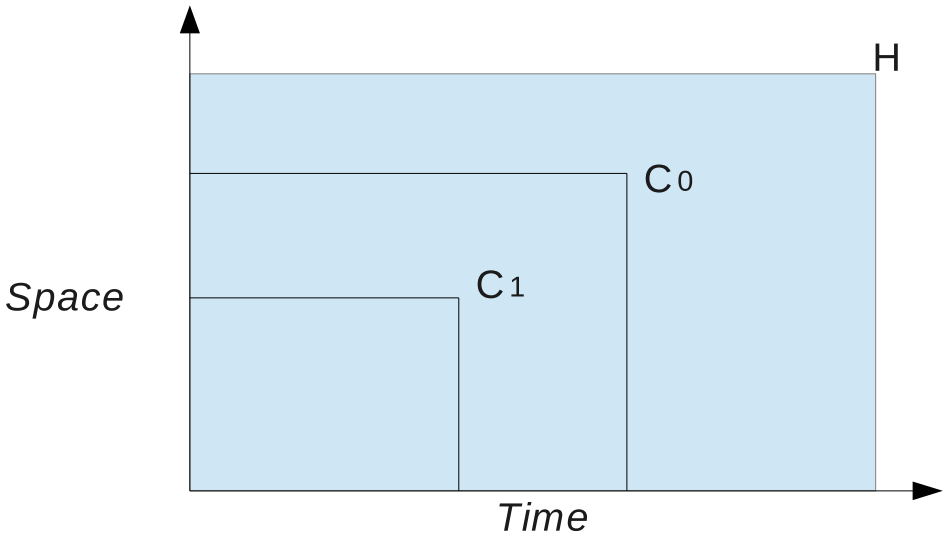


FIGURE 6.1: Reducing space and time: from a Haskell model to a C implementation (not to scale).

implementation, and a description of the main structure of our C implementation which we call *TyreCheck*C_0 . In §6.6 we measure the efficiency of *TyreCheck*C_0 . In §6.7 we improve the efficiency of *TyreCheck*C_0 , to obtain *TyreCheck*C_1 . In §6.8 we compare the Haskell model against the two C implementations and discuss the results. Finally, in §6.9 we present a brief discussion of the benefits of the implementation.

6.2 Space and Time Costs

As we saw in Chapter 2 one of the main goals of approaches such as proof-checking code is a minimal trusted computing base [5, 29]. The resources of memory and time are limited. We aim to provide an efficient way of checking the low-level code to prevent run-time errors. There are some alternatives to reduce the memory resources and improve the speed in TyreCheck. We might, for example, follow the approach of improving the performance in the data structures as in the work of Okasaki [92], redefining the Haskell data structures in a clever way. Another possibility is to translate our Haskell prototype to a C implementation. We have chosen the option of translating to a C version, in part because C provides scope for further low-level optimisations, and because we can take advantage of other imperative style properties, for instance, in-place updates of data type structures. In Figure 6.1 we depict this idea of *compressing* the time and space of the Haskell prototype denoted by `H`. We start by translating to the first version of C implementation denoted by $\texttt{C}_0$, then we improve the implementation leading to our second version of the implementation called $\texttt{C}_1$.

6.3 Space and Time of TyreCheckH

Before we commence, we give the size of the programs involved in the experiments of this Chapter. This will allow us to measure the behaviour of our tools in different scenarios for scalability testing. Table 6.1 shows the size of each program in terms of atoms and templates. The programs are in ascending order according to their number of atoms. In the following sections we will see how the time and space costs vary according to the number of atoms and the number of templates in a given program.

TABLE 6.1: Number of atoms and templates in programs involved in experiments for time and space cost.

Program	Atoms	Templates
MSS	144	37
Parts	147	34
Queens	157	30
OrdList	170	33
PermSort	172	36
Queens2	222	48
Braun	229	72
Taut	319	72
While	441	61
Cichelli	698	146
Sudoku	894	143

Table 6.2 gives us a general idea of the amount of memory allocated and time needed to infer the type information using the Haskell prototype. On average the allocated memory is 4Gb and the runtime is around 3.5 seconds. If we remove `Sudoku` from the experiments the average is considerably reduced to 2 Gb and 1.5 seconds respectively. According to Table 6.1 the number of atoms in `Sudoku` is more than four times the average number of atoms in the rest of the programs (which is 210), and the number of templates is almost four times the average numbers of templates in the other programs (44). We can ask the question: Is there any relation between the number of atoms or templates in the code for a program and the space-time used to compute the type information in the given set of programs?

In Figure 6.2 we have a graphical representation for the growth of space and time compared to atoms (a) and (b), templates in (c) and (d), number of atomic applications in (e) and (f), and the sum of atoms for functions, case tables, and constructors in (g) and (h).

In (a) and (b), we expected to see a pattern where the space and time increases when the number of atoms increases. But the variety of kinds of atoms makes this model too simple. Similarly in (c) and (d), we find that the number of templates is not a good predictor of space and time costs. One reason is because we can find *empty templates* (eg. consider the empty branch of a head function) or *single atom templates*.

TABLE 6.2: Summary of total allocated memory and time for Haskell prototype.

Program	Memory Allocation (Mb)	Time (s)
Parts	476.32	0.35
Queens	518.25	0.39
MSS	676.24	0.55
PermSort	761.37	0.58
OrdList	950.80	0.65
Queens2	1848.27	1.93
Braun	2209.6	2.26
Taut	2987.75	2.91
While	8129.82	7.23
Cichelli	15325.84	31.42
Sudoku	19970.88	26.55

Another alternative is to count the number of applications in each program. In (e) and (f), there is a closer correspondence between the number of applications and the space and time used. However, there still some inconsistencies. If we observe the plots (a) to (f) we can discard (c) and (d), the measures for templates. If the atoms and the applications are the hint, one possibility is only to take into account the most complex atoms. That is the atoms for functions, case tables and constructors. In Figure (g) and (h) we can see that the number of atoms of this kind is quite a good predictor of checking costs. Here we only take into account FUN, TAB and CON atoms in each program for the following reasons :

- The FUN atoms for functions are most often applied to some arguments, then, the presence of one atom FUN i j implies that there are possibly i atoms for arguments. Moreover, every time a function is called TyreCheck creates an instance or copy of the function type.

- The TAB atoms for case tables are similar, and more expensive in terms of memory. For instance the atom TAB i j k requires TyreCheck to copy the types for the templates at index position i to i+j. In addition, it must deal with any additional atoms after the TAB atom, representing the free variables denoted by k.

- The atom CON i j is a constructor of arity i and index j. The algebraic data types are related to this constructor. These atoms are very often applied to some i arguments.

The reason that atoms for primitives, pointers, literals and function arguments (PRI, PTR, INT and ARG) do not influence TyreCheck costs so much is that many of them are eliminated during PrimCheck analysis, some arguments become integer blocks, and some pointers are inlined. —See §4.3.

6.3.1 Profiling the Haskell Prototype

We shall now look more closely at *time* and *space* costs when we apply *TyreCheckH* to Reduceron code. The `ghc` compiler provides a mechanism to extract information about memory and time, and the *ghc profiler* [93, 94] collects the statistics. For now we execute the profiler for *TyreCheckH* applied to *Queens*, *Braun*, and *Sudoku* programs. From Table 6.1 we see that *Queens* has 157 atoms and 30 templates, *Braun* has 229 atoms and 72 templates, and *Sudoku* has 894 atoms and 143 templates.

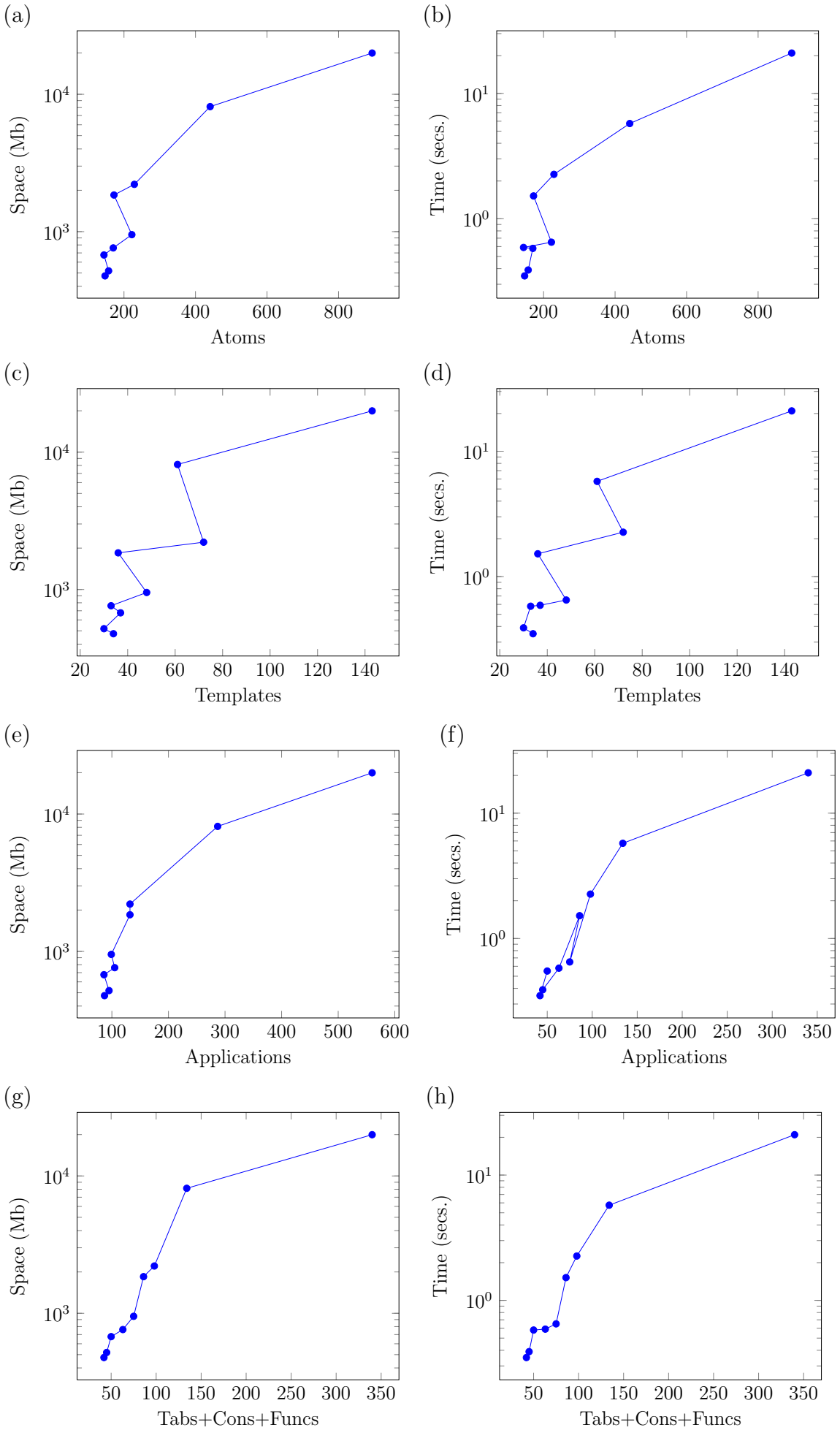
Figure 6.3 shows which TyreCheck functions consume most memory and time when we check the *Queens* code. In the Table the column `COST CENTRE` denotes the column for the name of the function. There are three functions that consume more than 83% of the time, and more than 84% of memory, these functions are:

- Function `freshvar` in module `Vars and TermOperations`, provide us a fresh term-type variables during type-checking. For the Haskell model we use a *brute force* style to obtain the next fresh variable. We traverse all the structures from the environment and the systems of equations to obtain a new fresh variable. In the C implementation we eliminate this memory consumption by using a global variable which gives us the next fresh variable without using the brute force solution used in Haskell prototype.

- Function `addDeclEnv.unkndefns` in module `Environment`, calculates which variables are *generic* in an environment. This operation is costly because it has to order the list of variables during its computation.

- Function `vars` in module `Terms`, collects a list of all the variables in a term-type expression. The accumulation of the lists is a big issue. These var lists in some cases must be in some order. With this restriction we compute expensive data structures over lists.

FIGURE 6.2: Code measures plotted against space and time costs.



COST CENTRE	MODULE	%time	%alloc
freshvar	Vars	28.1	52.9
addDeclEnv.unkndefns	Environment	20.6	1.2
vars	Terms	14.6	15.7
freshvar	TermOperations	7.4	7.1
freshvar	Vars	6.8	0.0
freshvar	Vars	6.5	5.8
unknownsScheme	Environment	3.2	7.3
vars.\	Terms	2.1	2.1
unknownsScheme.varsin	Environment	1.4	0.6
val	Environment	1.3	0.1
check	Main	1.1	0.4
vars	Terms	0.9	1.7
readPrec	Syntax	0.8	1.3

FIGURE 6.3: GHC time and allocation report for TyreCheckH for the program `Queens`.

What happens when we profile bigger programs like *Braun* and *Sudoku*? In Figures 6.4 and 6.5 the results are similar to the ones we obtained for *Queens*. The main functions taking almost all time and space are the same as in previous example. Additionally from the information in tables, we see that time used in computing the type information for *Braun* is 9 times the time used to compute the type information of *Queens*. And *Sudoku* takes more than 9 times the time of *Braun*. The most expensive functions remain the same here when we compute the type information for *Sudoku*. Notice the `freshvar` function in the three tables, it consumes more than 50% of memory in all the cases. The second most expensive function in terms of memory is the function `vars`. It collects the variable set for a term-type. In some cases, the name of the functions in column `COST CENTRE` are duplicated. For instance, the cost center `vars` belongs to a top level function, and `vars.\` belongs to a where clause in a function definition.

6.4 From TyreCheckH to TyreCheckC model

As a general rule, there is one module in the C implementation for each module in Haskell, one top level C function definition corresponds to a Haskell definition when possible, and there is a correspondence one-to-one for each data type structure.

COST CENTRE	MODULE	%time	%alloc
addDeclEnv.unkndefns	Environment	36.6	1.3
freshvar	Vars	23.1	52.4
vars	Terms	10.1	15.3
freshvar	TermOperations	6.3	7.2
freshvar	Vars	5.2	5.7
unknownsScheme	Environment	4.3	9.9
freshvar	Vars	4.1	0.0
val	Environment	2.3	0.1
vars.\	Terms	1.7	1.8
unknownsScheme.varsin	Environment	1.6	0.7
vars	Terms	0.8	1.8
vars	Terms	0.5	1.0

FIGURE 6.4: GHC time and allocation report for TyreCheckH for the program `Braun`.

COST CENTRE	MODULE	%time	%alloc
addDeclEnv.unkndefns	Environment	40.7	0.7
freshvar	Vars	22.1	56.4
vars	Terms	9.9	16.4
freshvar	TermOperations	6.3	7.9
freshvar	Vars	5.6	6.3
freshvar	Vars	4.8	0.0
unknownsScheme	Environment	2.3	5.9
vars.\	Terms	2.3	3.0
val	Environment	2.3	0.0
vars	Terms	0.5	1.1

FIGURE 6.5: GHC time and allocation report for TyreCheckH for the program `Sudoku`.

The different stages of checking Reduceron code are, *AtomCheck*, *PrimCheck*, and then *TyreCheck*. To explain the initial translation from Haskell to C, we give first, the translation approach for template code. Then we give a brief explanation of the other data structures used in the C implementation of TyreCheck.

6.4.1 Template Translation Overview

The structures in template code include, the structure of the atoms, applications and templates. For *PrimCheck* we also need split templates. We follow the

```
data Atom  =  ARG Int      | PTR Int      | INT Int | PRI String
            | FUN Int Int | CON Int Int | TAB Int Int Int
            | BOOL         | PRIM Int BOOL
```

FIGURE 6.6: Representing atoms in Haskell.

same pattern as in the Haskell prototype to represent the data structures in the C language implementation. For example, in Figure 6.7 we give the C translation of the Haskell declaration of the `Atom` data type in Figure 6.6. In general, we encode algebraic data types in Haskell as tagged unions in `struct` data type definitions. We can find an explanation how data types in Haskell can be translated to an imperative style in Appendix A of [95].

```
struct atom{
  enum {isArg, isPtr, isInt, isPri, isFun, isCon, isTab, isBool, isPrim} tag;
  union {
    int arg;
    int ptr;
    int num;
    PName pri;
    struct {int arity; int index;} fun;
    struct {int arity; int index;} con;
    struct {int index; int alts; int fvs;} tab;
    int bool;
    struct {int arity;int isBoolRes;} prim;
  };
};

typedef struct priname *PName;

struct priname{
 enum {Add, Sub, Leq, Equ, Dif, Emit, EmitInt} name;
};
```

FIGURE 6.7: Representing atoms in C.

Now that we have the definition for atom translation, to represent applications in C we need to encode Haskell lists. At first we use a translation really close to our Haskell prototype, defining a `head : tail` structure by using linked lists in our C implementation. Figure 6.8 give us the definition for a linked list representing a list of atoms. This is, a struct called `atomCons`, which has two components, the head of the list denoted `headAtom`, which is a pointer to a `struct atom` previously defined in Figure 6.7. The second component of the structure `struct atomCons`

```
typedef struct atom *Atom;
typedef struct atomCons *AtomList;

struct atomCons {
  Atom headAtom;
  AtomList tailAtoms;
};
```

FIGURE 6.8: Atom lists in C.

```
Atom mkARG(int v){
Atom a = (Atom) malloc(sizeof(struct atom));
a->tag = isArg;
a->arg = v;
return a;
}

AtomList mkAtomCons(Atom a, AtomList as){
AtomList al = (AtomList) malloc(sizeof(struct atomCons));
al->headAtom  = a;
al->tailAtoms = as;
return al;

}
```

FIGURE 6.9: Memory allocation constructors for atoms.

is the `tail` of the atom list, a recursive call to the linked list, this is encoded as `AtomList tailAtoms`. The empty list is represented by a `NULL` pointer.

To create an atom or a list of atoms, we have definitions such as those in Figure 6.9.

Finally, recall that the template code is represented in Haskell as in Figure 6.10. A program is a list of templates, and each template is a triple composed of arity, a list of atoms, and a list of list of atoms. The C translation is given in Figure 6.11.

The definitions given here are only illustrative examples of how we make the translation.

In Haskell the functionality for parsing and printing data structures is free. This is not the case in C but the techniques for parsing and pretty printing are straightforward and will not be discussed here.

```
type Arity    = Int
type Template = (Arity,[Atom],[[Atom]])
type Prog     = [Template]
```

FIGURE 6.10: Haskell template declaration.

```
typedef struct tplate *Template;
typedef struct templateCons *TemplateList;
typedef struct appCons *AppList;

struct appCons{
  AtomList headApp;
  AppList tailApps;
};

struct tplate{
  int arity;
  AtomList spineapp;
  AppList offspineapps;
};

struct templateCons{
  Template headTemplate;
  TemplateList tailTemplates;
};
```

FIGURE 6.11: Template definition in C.

6.4.2 Term-Types Translation Overview

Figure 6.12 follows the same approach of translating type-terms as we did in translating templates. The struct term gives us the support to work with type-term functions, type term variables, algebraic data types and numeric literals.

6.4.3 Remarks on C Data Structures

Beyond the representation of the terms, it is important to explain some considerations. Sometimes we can take advantage of in-place updates in imperative structures or loops instead of recursion. The data structures in Haskell are persistent, but in many cases their use is single threaded, in a C implementation we can

```
typedef struct term *Term;
typedef struct termCons *TermList;

typedef struct ics *Ics;
typedef struct icsCons *IcsList;

typedef struct ext *Ext;

typedef int Id;
typedef struct idCons *IdList;

struct term {
  enum {isFunT, isVar , isAlg, isNum} tag;
  union {
    struct {Term arg; Term res;} fun;
    int var;
    struct {IcsList ics; Ext xt;} alg;
    int num;
  };
};
```

FIGURE 6.12: An illustrative example of type-term in C.

```
Eqn drawEqnFor(int v, Sys s)
{
  EqnList *esref = &(s->eqnList);
    while (*esref) {
         Eqn e = (*esref)->headEqn;
            if (isVar(e->lhs) && e->lhs->var == v) {
                 *esref = (*esref)->tailEqns;
                 return e;
            }
         esref = &((*esref)->tailEqns);
    }
  return NULL;
}
```

FIGURE 6.13: The drawEqnFor function removes in-place any assignment found.

update in place. The system of equations used in the solver is a good example. The C version of the function `drawEqnFor` in Figure 6.13, destructively removes one equation from a system of equations stored in a linked list.

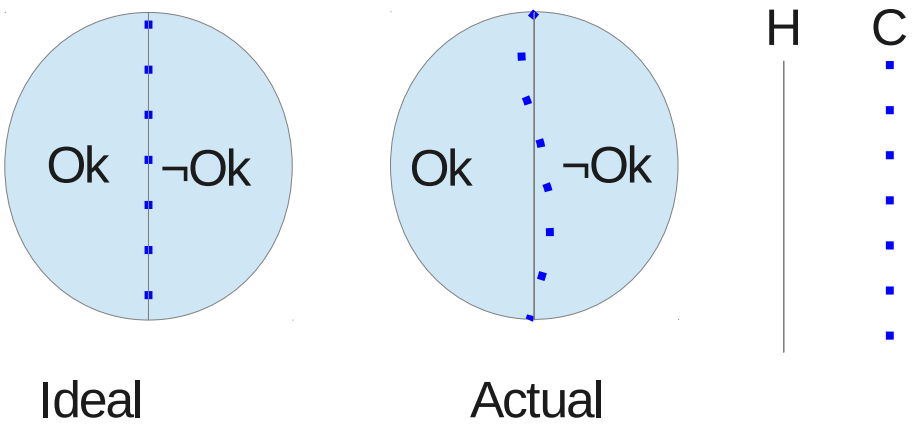


FIGURE 6.14: Interpretation diagram of compatibility results.

6.5 Correspondence Results

In this section we discuss the way to check the correspondence between the Prototype Checker implemented in Haskell and the checker implemented in C. Our *empirical* method is to mutate the programs of our benchmark, then we apply *TyreCheckH* and *TyreCheckC* to a large number of randomly generated *mutants*. If the *results are the same* for each mutant, then we have strong evidence of correspondence between the Haskell prototype and the C implementation.

From the set of programs presented in Table 6.1 in §6.3, we have executed the Haskell prototype and the C implementation against a set of mutations. The results are that in 99% of cases *results are the same*.

Some improvements in code produced the effect depicted In Figure 6.14. The ideal results are denoted by circle called `Ideal`; that means that all the OK programs are the same in the prototype and in the implementation, and all well-typed programs are the same in the implementation and in the prototype.

In the `Actual` circle we see the behaviour and that means that sometimes the programs stopped by the C implementation are not stopped by the Haskell prototype, or vice-versa. The C implementation is denoted by the dotted lines and the

template alternative of arity n : (n,[...],[...])
functional type in general : $a_0 \rightarrow ... \rightarrow a_n$
functional type for template alternative of arity 2 : $a_0 \rightarrow a_1 \rightarrow a_2$
functional type in detail : $a_0 \rightarrow ... \ a_n \rightarrow t \rightarrow fv_0 \rightarrow ... \rightarrow fv_n \rightarrow r$

FIGURE 6.15: Illustrative example to assign functional types based on the arity.

```
# of Template
1             (_,[... TAB 2 2 3,ARG 0],[...])
2             (2,[...],[..])
3             (2,[...],[..])
```

FIGURE 6.16: An small example of TAB constructor and its related alternatives.

Haskell prototype with the continuous line.

6.5.1 Discussion

During the translation from the model to the implementation, we found some problems in the prototype that were detected by the actual implementation.

As an illustrative example of such problems, let us consider the problem when we want to extract an element of a given list by using the standard Haskell functions *drop* and *take*. The analysis of the TAB i j k constructor is the one that deserves more care. This constructor has the following information: i is the index of the first alternative, j is the number of alternatives to be considered, and k is the number of free variables.

The functional type-term of an alternative is based in the arity of the template. In Figure 6.15, the template has arity n, and the functional type for this template is denoted as $a_0 \rightarrow ... \rightarrow a_n$, where a_0 is the first argument type and a_n is the type result. For instance, if we have a template of arity two the functional type-term is $a_0 \rightarrow a_1 \rightarrow a_2$. But, if this functional type represents a case alternative, we need to distinguish the term-variables that are before the untouchable table argument, and the variables after this table argument.

In Figure 6.16, we have that in template number one, the TAB constructor refers to the second template as the first alternative, and the last alternative is the template number three. The number of free variables for this table is one, as encoded in the TAB atom.

For example let us suppose we have in the second template vars = [a,b,c]. The formula to know how many variables we want to take from the alternative is :

arity - *number of free vars* - 1. In this example this is 2-3-1=-2. If we want to take a *negative* number of elements from a list, we obtain a valid result. Here, the application `take -2 [a,b,c]` is equals to an empty list []. This error was trapped during the translation to C, and was undetected in the Haskell prototype due to the nature of the definition of `take` function in Haskell. A similar problem occurs when we try to drop elements of a given list, and the number of elements is negative.

The discussion presented in this section points out the benefit of this approach to build programs. We were able to make several small improvements to the Haskell prototype based on comparative testing against our C implementation.

6.6 Time and Space Costs of TyreCheckC$_0$

6.6.1 Time

One of our main objectives is to reduce the amount of resources used by the checking tools. We would like checking techniques to be applicable even in *small* devices and without delaying execution unduly. The machinery to check if the code is well-behaved must be faster and less complex than the machinery to produce the code, eg. a compiler.

We apply the profiling tools to our first C implementation to examine runtime costs. In Tables 6.17 and 6.18 we present the main *time* measurements for *Queens* and *Braun* programs. We executed TyreCheckC$_0$ 100 times to emulate a sufficient number of profiling sample points to give a good estimate of which functions are the expensive ones. The 10 most costly functions consume more than 90% of time.

The meanings of the columns are as follows :

- *time* The percentage of the total execution time spent in this function.
- *cumulative seconds* The cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

% time	cumulative seconds	self seconds	calls	name
36.53	4.84	4.84	7180700	findId
30.19	8.84	4.00	8798700	concatVars
13.51	10.63	1.79	9860700	appendVars
10.23	11.99	1.36	25618700	findEqnFor
1.66	12.21	0.22	25054300	substInTerm
1.13	12.36	0.15	6113800	vars
0.72	12.45	0.10	187300	drawEqnFor
0.60	12.53	0.08	9860700	mkIds
0.49	12.60	0.07	19360770	emalloc
0.45	12.66	0.06	125600	varsEqs

FIGURE 6.17: Profile time for the 10 most costly functions when checking `Queens` executed 100 times using implementation TyreCheckC_0.

- *self seconds* The number of seconds accounted for by this function alone.
- *calls* The total number of times the function was called.
- *name* This is the name of the function.

In Table 6.17 we can see that function `findId`, `concatVars`, and `appendVars` are the three most expensive functions. In the Haskell prototype we can observe a similar behaviour in Figure 6.3 in §6.3. The first function lookup for an element in a given list of ints. The functions `concatVars` and `appendVars` collect variables in a list of terms, they preserve the order of the elements in the list. The function `findId` is used to find ids, for example in the solver when it computes lacks information. The functions to concatenate and append variables, are expensive because they traverses all the list, in order to attach in the tail the next list or element. Sometimes the big list is the one which is traversed. The order of the elements in a list sometimes is useful during type-checking. For instance consider when we need to determine which variables are free variables, which are the ones for the table argument and which the result type and the arguments in a case alternative.

The fourth most expensive function is `findEqnFor` which is only used by the solver.

In Table 6.18 we can see that the first ten functions are the almost the first ten functions in Table 6.17. Notice that the five most costly functions consumes more than 90% in of the execution time.

```
  %   cumulative   self
time   seconds   seconds    calls    name
47.74     18.55    18.55 22640600    findId
22.52     27.30     8.75 27602500    concatVars
13.31     32.48     5.17 29195800    appendVars
 5.62     34.66     2.19 76105400    findEqnFor
 2.47     35.62     0.96 74330600    substInTerm
 1.03     36.02     0.40 17582000    vars
 0.98     36.40     0.38  4534800    idElem
 0.71     36.68     0.28 36596500    substInEqn
 0.68     36.94     0.27   275800    drawEqnFor
 0.62     37.18     0.24   316100    substInSys
```

FIGURE 6.18: Profile time for the 10 most costly functions when checking `Braun` executed 100 times using implementation TyreCheckC_0.

In Tables 6.17 and 6.18 have detected which functions consumes the most time, and how they are called.

6.6.2 Space

In previous section we explored the time costs of TyreCheckC_0, here now we discuss the memory usage when we apply TyreCheckC_0 to Queens and Braun programs. By instrumenting every point the code where we allocate memory, we can extract which functions are using the most memory, and the total amount of memory allocated.

In Table 6.3 we present the 10 most expensive functions and the total amount of memory used when we compute the type information for `Queens` and `Braun` programs respectively. In both programs the functions presented in Table 6.3 use more than 94% of the overall memory used in each program. Here again, the functions related to ids are involved. For instance, the function `mkId` is used to build lists with ordered elements. The function `mkVar` is used to create vars in functions related to lists of vars, like `appendVars` or `concatVars`.

Recall in the previous §6.3 where we investigated the memory usage for *TyreCheckH* . In Table 6.3 we can see that the amount of allocated memory used by *Queens* is `518.25 Mb`. Compared to the amount of memory for *TyreCheck*C_0 applied to *Queens* (in Table 6.6) that amount is `2.58 Mb`. The amount of allocated memory

TABLE 6.3: Percentage of memory for the 10 most expensive functions used when checking Queens and Braun using TyreCheckC$_0$.

% of memory Queens	Function	% of memory Braun	Function
45.09	mkIds	56.52	mkIds
26.98	mkVar	19.68	mkVar
8.25	mkEqnCons	5.04	mkEqnCons
6.83	mkEqn	4.19	mkEqn
3.06	mkFun	3.83	mkFun
0.96	mkAlg	1.84	mkEnv
0.90	mkTermCons	0.92	mkTermSys
0.77	mkEnv	0.88	mkScheme
0.77	mkIcsCons	0.86	mkAlg
0.75	mkIcs	0.85	mkIdCons

94.36% of the total memory used(2.58 Mb) in Queens.
94.60% of the total memory used(6.99 Mb) in Braun.

in Haskell prototype is *265 times* bigger than the memory used in C implementation for *Queens*. Similarly, for the experiments when we apply TyreCheckH and TyreCheckC$_0$ to *Braun*, the amount of memory in Haskell prototype is *307 times* the amount used in C implementation. Figure 6.19 shows the gain in space and time when we translate the Haskell prototype to a C implementation.

6.7 Space and Time of TyreCheckC$_1$

6.7.1 Time of TyreCheckC$_1$

From the observation in the data in preliminary sections Haskell prototype and the C implementation, we can conclude that functions involving lists of variables or a list of ids are the most used. Some of them are part of the solver based on the work of Colmerauer [19]. We can tackle the problem by using an efficient approach to solve infinite trees as in [96, 97], in which the searching of an equation during the substitution is performed in constant time.

In our first implementation in C, we emulate the list approach in Haskell, by using linked list in C. One advantage of this approach is that we follow the Haskell

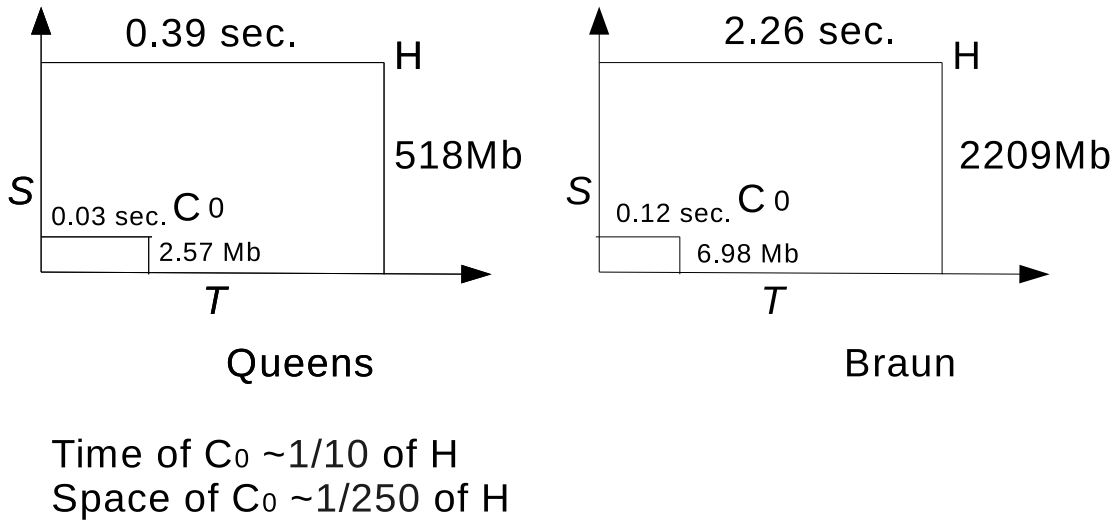


FIGURE 6.19: Comparative for space and time of TyreCheckC$_0$ and TyreCheckH.

```
 %   cumulative   self
time   seconds   seconds    calls    name
54.70      2.39     2.39  7180700    findId
20.25      3.28     0.89 25618700    findEqnFor
 6.18      3.55     0.27   555300    appendVars
 5.26      3.78     0.23 25054300    substInTerm
 1.60      3.85     0.07  6109000    vars
 1.37      3.91     0.06    81000    nubIdList
 0.92      3.95     0.04   949200    idElem
 0.80      3.98     0.04 19382270    emalloc
 0.80      4.02     0.04  8127400    fromVar
 0.69      4.05     0.03  7753500    mkIds
```

FIGURE 6.20: Profile time for the 10 most costly functions when checking `Queens` executed 100 times using implementation TyreCheckC$_1$.

prototype. The drawback of the linked-lists approach is that the operation of searching and updating in a lists is expensive. One possible way to reduce the time in searching to a constant time, is by using arrays.

% time	cumulative seconds	self seconds	calls	name
60.38	10.19	10.19	22640600	findId
13.95	12.55	2.36	76105400	findEqnFor
5.04	13.40	0.85	74330600	substInTerm
3.79	14.04	0.64	1248300	appendVars
1.78	14.34	0.30	4534800	idElem
1.72	14.63	0.29	17569600	vars
1.24	14.84	0.21	936000	appendEnv
1.07	15.02	0.18	316100	substInSys
1.07	15.20	0.18	439200	nubIdList
1.04	15.37	0.18	275800	drawEqnFor

FIGURE 6.21: Profile time for the 10 most costly functions when checking `Braun` executed 100 times using implementation TyreCheckC_1.

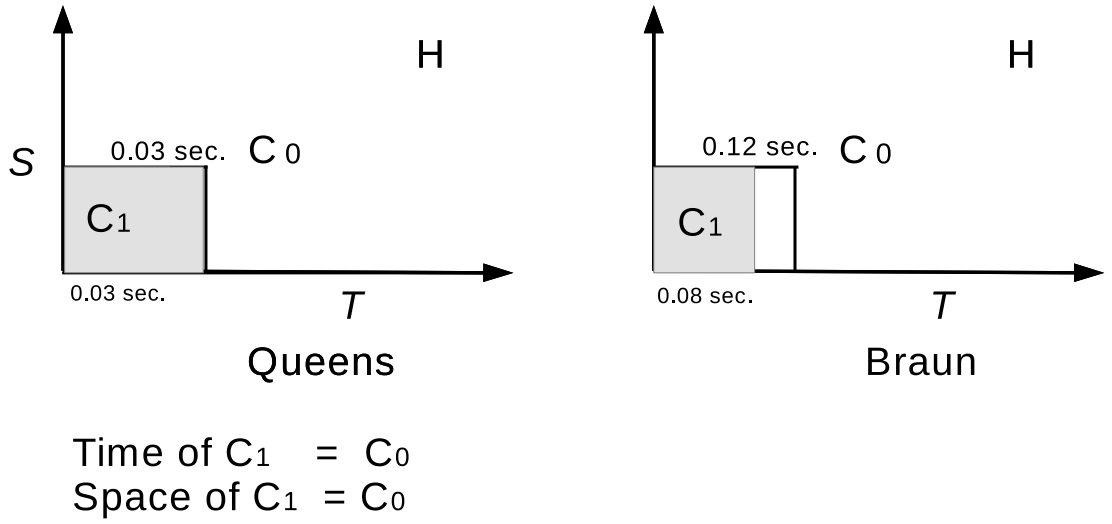


FIGURE 6.22: Comparative for space and time of TyreCheckC_1,TyreCheckC_0 and TyreCheckH.

TABLE 6.4: Percentage of memory for the 10 most expensive functions used when checking Queens and Braun using TyreCheckC$_1$.

% of memory Queens	Function	% of memory Braun	Function
35.42	mkIds	44.19	mkIds
26.95	mkVar	19.65	mkVar
10.33	mkIdCons	13.13	mkIdCons
8.24	mkEqnCons	5.04	mkEqnCons
6.83	mkEqn	4.19	mkEqn
3.06	mkFun	3.82	mkFun
0.96	mkAlg	1.84	mkEnv
0.90	mkTermCons	0.92	mkTermSys
0.77	mkEnv	0.88	mkScheme
0.77	mkIcsCons	0.86	mkAlg

94.21% of the total memory used (2.57 Mb) in Queens.
94.51% of the total memory used (6.98 Mb) in Braun.

6.7.2 Space of TyreCheckC$_1$

Even if we have sped up the time, we still have the same memory usage. In Table 6.4 we have the functions consuming more than 94% of the total amount of used memory. In Figure 6.22 we can notice the reduction in time. The second C implementation is about *2*× times faster than the first C implementation in the big programs. The memory used still almost the same, we can see it in the graphical representation of Figure 6.22.

Differences between TyreCheckC$_0$ and TyreCheckC$_1$. The implementation TyreCheckC$_1$ differs from TyreCheckC$_0$ in the following:

TyreCheckC$_1$ uses a clever way when concatenate or append lists of term variables. Instead of traversing the bigger list and adding it at the end the small list. We solve this problem by the use of continuation lists to avoid append applications. This operations on lists are performed when the elements of the accumulated list do not need to be in specific order.

6.8 Comparative Performance

Execution time We first present and discuss a comparison of execution times between the two C implementations and the Haskell model. The testing was performed using the `-O2` optimisation flag in `ghc` and `gcc` compilers.

In Table 6.5 we see the reduction in execution time from the model TyreCheckH to the implementation TyreCheckC0. This gain is about *10*×, except for the programs `While` and `Sudoku`, in which we only have a reduction in speed of about *3*× to *4*×. The reduction in execution time from TyreCheckH to TyreCheckC0 is in part because in the C implementation we use a counter to generate fresh variables. In Haskell version we use a brute force method to determine a fresh variable. The programs `Sudoku` and `While` demand more work in functions to extract term variables from the environment. Every time we have a new solution for a connected group, in the Haskell version we store only the needed variables and then we store them into the environment. In TyreCheckC0 we decided to omit this step of filtering needed variables, so the number of equations stored in the environment for each top-level function definition is bigger in the C implementation. The added cost becomes significant in programs where more complex structures are stored in the environment as for `Sudoku` and `While`.

When we use the TyreCheckC1 implementation, we see that for all but the smallest programs it is about *2*×–*3*× faster than the TyreCheckC0 implementation. The time needed to compute the type information in bigger programs is less than a second in the case of `While` (441 atoms, 61 templates) and `Cichelli` (698 atoms, 146 templates), and less than three seconds for `Sudoku` (894 atoms, 143 templates).

Memory allocation In Table 6.6 we have a comparison of memory allocation for the two C implementations and the Haskell model. The memory allocation is reduced by a factor of *200*× to *350*× when we use TyreCheckC0 instead of the model TyreCheckH. This saving is mainly due to the use of in-place update of the major single threaded data structures in the C implementation.

For TyreCheckC1 there is no further reduction in allocated memory in comparison with TyreCheckC0. The use of continuation lists in append applications avoids the cost of traversing the first append argument, but as append in TyreCheckC0

Table 6.5: Summary of *time* in seconds for the Haskell model and the C implementations.

Program	TyreCheckH	TyreCheckC0	TyreCheckC1
Parts	0.35	0.03	0.02
Queens	0.39	0.03	0.03
MSS	0.55	0.04	0.04
PermSort	0.58	0.03	0.02
OrdList	0.65	0.08	0.04
Queens2	1.93	0.26	0.11
Braun	2.26	0.12	0.08
Taut	2.91	0.26	0.11
While	7.23	2.94	0.99
Sudoku	26.55	6.05	2.74
Cichelli	31.42	2.23	0.87

Table 6.6: Summary of *allocated memory* in MB for the Haskell model and the C implementations.

Program	TyreCheckH	TyreCheckC0	TyreCheckC1
Parts	476.32	2.66	2.66
Queens	518.25	2.58	2.57
MSS	676.24	2.88	2.87
PermSort	761.37	3.07	3.06
OrdList	950.80	3.65	3.64
Queens2	1848.27	7.44	7.44
Braun	2209.60	6.99	6.98
Taut	2987.75	10.14	10.13
While	8129.82	25.53	25.52
Cichelli	15325.84	38.94	38.92
Sudoku	19970.88	59.07	59.05

already uses destructive update no further reduction in allocated memory is possible.

A Really Fast Implementation From the experiments reported in this section, we can see that the execution time needed to check bigger programs using TyreCheckC1 is still 1-3 secs. To have a really fast implementation we would need to change the representation of some heavily used data structures. For instance, at some points the solver needs to look for an equation with a specific variable on

the left. The access to any such equation could be achieved in *constant* time if we used an array structure with variables as indices. In order to know how much memory we need to allocate such arrays, we can simply count the number of type variables.

In our experiments in this thesis, the representation for the Reduceron code uses linked date structures. However, the actual machine executes Reduceron template code represented as a blocks of bytes. So this represents another opportunity for optimisation, the template code could also be represented with indexed addressing to have the benefits of constant time access.

6.9 Summary

In this Chapter we have shown how to reduce the space and time costs of TyreCheck.

The main benefit of having a small and fast checker is to provide an efficient way of checking Reduceron code without relying too much on the computational power of the target machine where our static methods are applied.

We started with a Haskell prototype, which served as a basis for implementations in C. Then in the first stage We build TyreCheckC_0 implementation and measure it. We extracted information about where the time is going and how much memory is used in each program allocation point. In TyreCheckC_1, the second C implementation we have a good gain in the performance. We have reduced the memory from TyreCheckH to TyreCheckC_0 *300*$\times$ in average for test programs. In the second version, for TyreCheckC_0 to TyreCheckC_1 we have a gain in speed is about *2*$\times$.

Further optimisations might lead to better results than the ones presented here. A further analysis of the data structures used in the implementation is a good start. For instance, to reduce the amount of memory and increase the speed we can calculate the maximum number of allocated variables in all the programs. Once we have that information, we can change the data structure representation to arrays. With that number of the possible allocated variables, we can allocate memory for the entire array of variables. By using arrays we can access the type-term variables in constant time, those variables are references to other type of term-types structures like functions or algebraic data types. Another benefit of

having arrays, is that we can manipulate the memory possibly in blocks of memory or blocks of related variables.

Chapter 7

Conclusions

7.1 Summary of Contributions

This thesis has explored and discussed type-checking methods applied to low-level code. In Chapter 4 we developed type-checking methods to check statically low-level code. In Chapter 5 types derived from low-level code were shown compatible with types inferred from the high-level source program. Chapter 6 showed the first steps towards efficient static checkers.

In Chapter 2 we presented a literature review of some related work, noting connections between what we explore in this thesis and what has been reported in the literature. The whole area of static checking is big, we haven't discussed all the techniques in our work. In our work we have focussed more on lightweight static checking techniques, particularly those related to type-checking and type-inference.

The main contribution of Chapter 3 is the machinery to test our prototypes and implementations. The main idea is based on *mutations*. The mutation of programs gives us a convenient way to create programs very close to a compiled and well-behaved program. We combine mutation with the random selection of the mutated candidate.

In Chapter 4 we proposed a method based on type equations to infer type information from low-level code. In addition, we have *evaluated the effectiveness* of the static checking techniques by *empirical work* – see §4.5. We have used the mutation technique to generate mutants based on a well-behaved program.

Also in Chapter 4 we proposed a method to eliminate the primitive applications in a preliminary checking stage called *PrimCheck* , see §4.3. Instead of dealing with dependent types, we reduce and isolate the problem. We measure the effectiveness of the primitive checking alone by using mutations. The advantage of PrimCheck is that we can stop bad programs before type-checking them with fewer resources in terms of computation steps and memory usage.

In Chapter 5 we provided evidence of compatibility between the type inference system for a high-level and the one used by the checkers working on low-level code. We showed how to translate high-level types to low-level type-terms, then we can compare or *solve* the two systems of equations. The initial idea is if we can find a joint solution then there is a *compatibility* between the two type systems. We also check some other properties relating solutions of the two systems.

The final part of this thesis is about the time and space costs of our static checker. In Chapter 6, we created a C version of the Haskell prototype to make more efficient static checker, in terms of memory consumption and execution time. The first part is concerned about the correspondence between the Haskell prototype and the C implementation. Some updates over the structures are performed in place rather than producing a new structure as it is in functional programming languages. The second part is more related to the efficiency of the C implementation. We have measured and detected the most costly parts of the code. A systematic transformation eliminates some of the most costly list processing, and this transformation alone yields a 3× speed up.

7.2 Discussion

The importance of the checking low-level code is essential in a connected world. The compiled code may be sent over an unreliable channel, and could be altered to produce an unexpected behaviour in the host machine. The need to patch compiled code in a running system is part of the reality in the software industry. We have experimented with the combination of functional languages and type-checking techniques to create a prototype, and then implemented a checker in C language to check Reduceron code statically.

The advantage of the static checking techniques presented in our work is significant, because we do not need complex machinery such as theorem provers. No extra

load is placed on the programmer or on the compiler. The programmer does not need to learn how to use a theorem prover or install new features into the compiler. The compiler does not need to be adapted to a theorem prover. We have provided a mechanism *to check statically low-level untyped code.*

7.3 Future Work

The work could be developed further in several directions. Possible lines of future work include :

- *By adding a small amount of code annotation the effectiveness of the static checking might be improved.* Even if most ill-behaved programs can be trapped by the static analysis, and most well-behaved programs accepted, by adding some extra information the effectiveness might be improved. This extra information is not just type information, some other properties can be useful to help in the stopping of ill-behaved programs and allowing more well-behaved programs. Some related work on annotating programs is found in [13–15]. A combination of type-checking and property checking by adding annotations as in [16], could provide a more powerful mechanism to stop ill-behaved programs.

- *In the case of adding a new component, checking would ideally be applied only to the specific component, avoiding the extra cost and complexity of whole-program checking.*

  This static checking mechanism could be applied to a part of whole program. In the same spirit of functional programming with a strong type system, we could use the type information to check if the new component is type compatible with the existing type information in the context.

- We have not investigated the properties of the mutation test, this is a whole area and we have taken only the basic principles for our experimental work. We think this model of testing is general enough to be used in other experiments.

Bibliography

[1] Matthew Naylor and Colin Runciman. The Reduceron Reconfigured. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*, pages 75–86, 2010.

[2] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987. ISBN 013453333X.

[3] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice-Hall, Inc., 1992. ISBN 0-13-721952-0.

[4] D. L. Clutterbuck and B. A. Carré. The Verification of Low-level Code. *Softw. Eng. J.*, 3(3):97–111, 1988. ISSN 0268-6961.

[5] George C. Necula. Proof-Carrying Code. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, 1997.

[6] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, 21(3): 527–568, 1999.

[7] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-level Programming. In *Proc. 16th European Conference on Programming (ESOP'07)*, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Simon Winwood and Manuel Chakravarty. Singleton: A General-Purpose Dependently-Typed Assembly Language. In *Proc. 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*, pages 3–14, 2011.

[9] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic Foundations for Typed Assembly Languages. *ACM Trans. Program. Lang. Syst.*, 32:7:1–7:67, 2010.

[10] David Evans. Static Detection of Dynamic Memory Errors. In *Proc. of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, PLDI '96, pages 44–53, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2.

[11] Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, and Germán Puebla. Verification of Java Bytecode Using Analysis and Transformation of Logic Programs. In *Proceedings of the 9th international conference on Practical Aspects of Declarative Languages*, PADL'07, pages 124–139, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-69608-3, 978-3-540-69608-7.

[12] Philip W. L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems*. PhD thesis, Simon Fraser University, Burnaby, BC, Canada, Canada, 2004.

[13] Yann Régis-Gianas and François Pottier. A Hoare Logic for Call-by-Value Functional Programs. In *Proc. of the International Conference on Mathematics of Program Construction*, MPC '08, pages 305–335, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70593-2.

[14] Bernhard Möller. Applicative Assertions. In *Proc. of the International Conference on Mathematics of Program Construction*, pages 348–362, London, UK, 1989. Springer-Verlag. ISBN 3-540-51305-1.

[15] Tom Schrijvers, Louis-Julien Guillemette, and Stefan Monnier. Type Invariants for Haskell. In *Proc. of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 39–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-330-3.

[16] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying Type Checking and Property Checking for Low-Level Code. *SIGPLAN Not.*, 44(1):302–314, January 2009. ISSN 0362-1340.

[17] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[18] Luca Cardelli. *The Computer Science and Engineering Handbook*, chapter 103, Type Systems, pages 2208–2236. CRC Press, 1997. ISBN 0-8493-2909-4.

[19] A. Colmerauer. *Prolog and Infinite Trees. Logic Programming*, pages 231–252. Academic Press, 1982.

[20] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1.

[21] Raymie Stata and Martn Abadi. A Type System for Java Bytecode Subroutines. In *In Proceedings of the 25th ACM POPL*, pages 149–160, 1998.

[22] C. Pusch. *Formalizing the Java Virtual Machine in Isabelle-HOL*. Inst. für Informatik, 1998. URL http://books.google.co.uk/books?id=hw26GwAACAAJ.

[23] R. Cohen. The Defensive Java Virtual Machine Specification. Technical report, Computational Logic inc., 1997.

[24] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *TCS*, 298: 583–626, 2003.

[25] Xavier Leroy, Inria Rocquencourt, and Trusted Logic S. A. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30:2003, 2003.

[26] 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 *In Second Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[27] Andrew W. Appel. Foundational Proof-Carrying Code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247. IEEE Computer Society, 2001.

[28] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A Provably Sound TAL for Back-end Optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 208–219, 2003. ISBN 1-58113-662-5.

[29] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational Proof Checkers with Small Witnesses. In *PPDP '03: Proceedings of the 5th ACM*

SIGPLAN International Conference on Principles and Practice of Declarative Programming, pages 264–274, 2003.

[30] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. *Electr. Notes Theor. Comput. Sci.*, 132(1):113–129, 2005.

[31] Elvira Albert, Puri Arenas, Germán Puebla, and Manuel V. Hermenegildo. Certificate Size Reduction in Abstraction-Carrying Code. *TPLP*, 12(3):283–318, 2012.

[32] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. Abstraction-Carrying Code: a Model for Mobile Code Safety. *New Generation Comput.*, 26(2):171–204, 2008.

[33] Y Jia and M Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[34] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[35] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, 1969.

[36] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975.

[37] Sidney L. Hantler and James C. King. An Introduction to Proving the Correctness of Programs. *ACM Comput. Surv.*, 8(3):331–353, September 1976. ISSN 0360-0300.

[38] Magnus O. Myreen and Michael J. C. Gordon. Hoare Logic for Realistically Modelled Machine Code. In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), LNCS*, pages 568–582, 2007.

[39] Magnus O. Myreen. Formal verification of machine-code programs. Technical Report UCAM-CL-TR-765, University of Cambridge, Computer Laboratory, December 2009. URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-765.pdf.

[40] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. ISSN 0360-0300.

[41] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000. ISSN 1388-3690.

[42] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[43] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[44] Simon Peyton-Jones. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, 2003. ISBN 9780521826143.

[45] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55. ACM, 2007. ISBN 978-1-59593-766-7.

[46] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32:98–107, 1984.

[47] Roberto Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 15(4):575–631, 1993.

[48] R. Stansifer. Type Inference with Subtypes. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 88–97. ACM, 1988.

[49] John C. Mitchell. Type Inference With Simple Subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[50] Stefan Kaes. Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *LISP and Functional Programming*, pages 193–204, 1992.

[51] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient Recursive Subtyping. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 419–428. ACM, 1993.

[52] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive Subtyping Revealed. In *Journal of Functional Programming*, page 2003, 2000.

[53] Jens Palsberg and Tian Zhao. Type Inference for Record Concatenation and Subtyping. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 125–136. IEEE Computer Society Press, 2002.

[54] Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. Typed Compilation of Recursive Datatypes. In *In ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI*, pages 98–108, 2003.

[55] Dario Colazzo and Giorgio Ghelli. Subtyping, Recursion and Parametric Polymorphism in Kernel Fun. *Inf. Comput.*, 198(2):71–147, May 2005.

[56] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

[57] Zhenyu Qian. A Formal Specification of Java Virtual Machine Instructions. In *Formal Syntax and Semantics of Java TM . Springer Verlag LNCS*, pages 271–311. Springer-Verlag, 1997.

[58] James Gosling. Java Intermediate Bytecodes: ACM SIGPLAN Workshop on Intermediate Representations (ir'95). In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 111–118. ACM, 1995.

[59] M. Kaufmann and J S. Moore. The ACL2 Home Page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2004.

[60] Gerwin Klein and Martin Strecker. Verified Bytecode Verification and Type-certifying Compilation. *Journal of Logic and Algebraic Programming*, 58(1–2): 27–60, 2004.

[61] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003. URL http://www4.in.tum.de/~kleing/diss/.

[62] G. Klein and M. Wildmoser. Verified Bytecode Subroutines. *J. Autom. Reasoning*, 30:363–398, 2003.

[63] Xavier Leroy. Bytecode Verification on Java Smart Cards. *Software Practice and Experience*, 32:2002, 2002.

[64] Eva Rose. Lightweight Bytecode Verification. *J. Autom. Reasoning*, 31(3-4): 303–334, 2003.

[65] Karsten Klohs and Uwe Kastens. Memory Requirements of Java Bytecode Verification on Limited Devices. *Electron. Notes Theor. Comput. Sci.*, 132 (1):95–111, May 2005.

[66] Sonia Fagorzi and Elena Zucca. A Framework for Type Safe Exchange of Mobile Code. In *in: TGC 2006 - 2nd International Symposium on Trustworthy Global Computing 2006, LNCS*, 2007.

[67] Heidar Pirzadeh. Encoding the Program Correctness Proofs as Programs in pcc Technology. In *PST '08: Sixth Annual Conference on Privacy, Security and Trust*, 2008.

[68] William Enck, Machigar Ongtang, and Patrick Mcdaniel. On Lightweight Mobile Phone Application Certification. In *ACM Conference on Computer and Communications Security*, pages 235–245. ACM, 2009.

[69] Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas Jensen, and David Pichardie. The MOBIUS Proof Carrying Code Infrastructure. In *Lecture Notes in Computer Science*, volume 5382, pages 1–24. Springer-Verlag, 2008.

[70] Gilles Barthe, Lennart Beringer, Pierre Crgut, Benjamin Grgoire, Martin Hofmann, Peter Mller, Erik Poll, Germn Puebla, Ian Stark, and Eric Vtillard. Mobius: Mobility, Ubiquity, Security: Objectives and Progress Report. In *Proceedings of the 2nd International Conference on Trustworthy Global Computing*, pages 10–29. Springer-Verlag, 2007.

[71] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying Type Checking and Property Checking for Low-level Code, 2009.

[72] Matthew Naylor and Colin Runciman. The Reduceron Reconfigured and Reevaluated. *J. Funct. Program.*, 22(4-5):574–613, 2012.

[73] F-lite: a core subset of Haskell. www.cs.york.ac.uk/fp/reduceron/memos/Memo9.txt, 2008.

[74] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In *In Trends in Functional Programming, volume 7. Intellect*, pages 157–172, 2007.

[75] Richard Hamlet. Random Testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[76] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed Random Test Generation. In *In ICSE*. IEEE Computer Society, 2007.

[77] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000.

[78] Koen Claessen and John Hughes. Testing Monadic Code with Quickcheck. In *IN PROC. ACM SIGPLAN WORKSHOP ON HASKELL*, pages 65–77, 2002.

[79] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[80] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation Analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, September 1979.

[81] James M. Bieman, Daniel Dreilinger, and Lijun Lin. Using Fault Injection to Increase Software Test Coverage. In *IN PROC. 7TH INT. SYMP. ON SOFTWARE RELIABILITY ENGINEERING (ISSRE'96*, pages 166–174, 1996.

[82] A. Jefferson Offutt and Stephen D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20:337–344, 1994.

[83] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering*, pages 5–99, 1996.

[84] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2.

[85] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

[86] A. Jefferson Offutt. Investigations of the Software Testing Coupling Effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992. ISSN 1049-331X.

[87] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In *IN PROC. ICSE*, pages 351–360, 2008.

[88] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. Quickchecking Refactoring Tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, pages 75–80, 2010.

[89] Jukka Paakki. Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[90] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, 2007. ISBN 978-1-59593-811-4.

[91] Mark P. Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell, 1999.

[92] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

[93] Ghc profiling. http://www.haskell.org/ghc/docs/7.2.1/html/users_guide/profiling.html. Accessed: 2013-09-13.

[94] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. ISBN 0596514980, 9780596514983.

[95] Simon Thompson. *The Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999. ISBN 0201342758.

[96] Joxan Jaffar. Efficient unification over infinite terms. *New Generation Computing*, 2(3):207–219, 1984. ISSN 0288-3635.

[97] Kuniaki Mukai. A unification algorithm for infinite trees. In *Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 1*, IJCAI'83, pages 547–549, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.