# Languages for Secure Multiparty Computation and Towards Strongly Typed Macros

Janus Dam Nielsen

PhD Dissertation


Department of Computer Science
University of Aarhus
Denmark

# Languages for Secure Multiparty Computation and Towards Strongly Typed Macros

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
PhD Degree

by
Janus Dam Nielsen
February 24, 2009

# Abstract

We show that it is feasible and useful to create programming languages with strong security guarantees for secure multiparty computation. We have designed and implemented the Secure Multiparty Computation Language (SMCL), which is a domain-specific programming language for secure multiparty computation. SMCL allows programmers to write programs using secure multiparty computation without expert knowledge on how to design and implement cryptographic protocols. We have proven that programs written in SMCL are immune to a broad range of security threads and confidential information may only be revealed in specific parts of a program, designated by the programmer.

We demonstrate the usefulness of SMCL by reporting on how an SMCL program contributed to the first large-scale practical application of secure multiparty computation.

Based on our experiences with SMCL we have designed a successor called PySMCL which is a domain-specific language embedded in Python, and will provide even better security guarantees than SMCL.

We also show that syntactic abstraction can enable programming languages to evolve over time by adding new libraries with syntax and semantics as they become needed. This is an important feature because it is difficult to anticipate which language concepts becomes needed over the lifetime of a programming language. We have designed and implemented a hygienic syntactic abstraction system for the Fortress programming language based on parsing expression grammars. The system allows syntactic extensions of Fortress to be specified in a modular fashion and added in libraries as needed. The system also allows us to not only support new language extensions, but also to move many constructs of the core language into libraries. We have designed a type system which extend the guarantees provided by the Fortress type system to the combined language of Fortress and our system for syntactic abstraction.

# Acknowledgments

First of all, I would like to thank Michael I. Schwartzbach for supervising my PhD and fruitful collaboration.

A thanks to my dear girlfriend Annette Friis Eriksen, for her invaluable support, patience, understanding, encouragement, and love during the last three years. Thank you for the many hours of fun and adventures we have had together, and thanks for listening the many times I have talked about my work.

I want to thank my mother Susanne Dam Nielsen, my brother Jens Verner Dam Nielsen, and my grandfather Verner Dam Nielsen for love and solid support. I am also deeply grateful for the support of my friends, especially Søren Reinertsen and Jens Kristian Uhrenholdt for being the outstanding friends they are, for being there when I needed it, and for countless hours of fun.

Thanks to the Fortress team for making my visit at Sun Microsystems possible, inspiring, and showing me a different view of research. Especially thanks to Eric Allen for being an outstanding host during my stay in Austin and for making it a memorable experience. And also thanks to Sukyoung Ryu, Jon Rafkind, and Ryan Culpepper for pleasant, enjoyable, and successful cooperation, it has been fun to getting to know you. Also a thanks to my colleagues on the SIMAP and CACE projects for good collaboration and fun discussions.

Thanks to my office-mates during the years, Martin Mosegaard Jensen, Aske Simon Christensen, Christian Kierkegaard, Johnni Winther, and Claudio Orlandi for fun discussions and entertaining distractions. And thanks to my colleagues Peter Mechlenborg, Mathias Romme Schwartz, Jakob Grauenkjær Thomsen, and Simon Holm Jensen for many interesting and inspiring discussions and fun distractions (especially the (almost) infinite number of table-soccer matches we have played). A special thanks to Johnni Winther for many enlightening conversations, I have learned so much.

Thanks to the administrative staff and the technical staff here at DAIMI. Especially thanks to Dorthe Haagen Nielsen and Ellen Kjemtrup Lindstrøm for many enlightening discussions, and to Kai Birger Nielsen, Rune Bolding Brobjerg, Michael Glad, and Jonathan Frumer for answering my many questions about the system setup.

*Janus Dam Nielsen,*
*Århus, February 24, 2009.*

# Contents

# Introduction

*Anything I've ever done that ultimately was worthwhile...*
*initially scared me to death*
— Betty Bender

In this dissertation we investigate whether it is feasible and useful to create programming languages with strong security guarantees for secure multiparty computation. We also investigate to what extend syntactic abstraction can enable programming languages to evolve over time by providing new syntax and semantics as they become needed.

Information is a resource of vast importance and economic value to individuals, public administration, and private companies, who spend more and more time and effort to protect the confidentiality of their information from other parties. Examples include the definition of security policies, access control (physical as well as electronic), careful destruction of old data, encryption of information, restrictions on allowed programs on company computers, and the wide spread use of anti virus programs and firewalls.

However, there are many situations where significant value can be obtained by combining confidential information from various sources (e.g. financial benchmarking, auctions, negotiations, and profile matching). In economics, it is well-known that the most precise market clearing price (the price at which supply meets demand) for auctions can be found if all participants reveal how they truly value the auctioned commodity. The information can be provided as a bid-strategy which is a function of the price describing how much of the commodity to buy at each price. However, the bid-strategy is sensitive data because it may be misused by a dishonest auctioneer. The auctioneer could collude with the seller and set the market clearing price to the maximum value anyone is willing to bid. This is a real world problem with the many online providers of auctions where the user submit a bid-strategy in terms of a maximum bid. The auctioneer now has the possibility of raising the bids on an item to the maximum of all maximum bids, which is usually not what the users expects or wants. Another example is profile matching. A buyer would find it useful to get recommendations on which music album to purchase without revealing the entire content of his/hers music collection, including any possibly embarrassing content. Yet another example is financial benchmarking where a number of companies have a common interest in computing a ranking of the companies, e.g. based on their individual productions costs in order to assess possibilities for reduction of production costs. However, none of the companies

are interested in revealing their own exact production costs to the others, because it might reveal valuable information about profit margins and robustness of the company.

Overcoming this fundamental conflict between the benefits of confidentiality and the benefits of information sharing is possible using the technique of secure multiparty computation. The technique has been available for quite some time. However, we have only recently begun to see real world application of secure multiparty computation, in fact we will report on how one of the programming languages described in this thesis contributed to the very first large-scale practical application of secure multiparty computation.

We release some of the yet unused potential of secure multiparty computation by providing domain-specific programming languages [99] with strong security guarantees for secure multiparty computation. The languages will allow programmers with a very limited understanding of secure multiparty computation to implement, e.g. secure versions of auctions and financial benchmarking using secure multiparty computation. In this way we enable the easy development of new applications which exploit confidential information without revealing it, to solve problems of considerable economic value.

Programming languages are generally designed to enable the programmer to express ideas clearly and concisely. But the set of concepts needed over the lifetime of a programming language is difficult to anticipate and is often dependent on the development of new systems that programs must interact with (for example, new domain-specific languages, new programming platforms, new virtual machines, etc.). If a programming language can adapt and support new concepts during the lifetime of the language it will retain its ability to express ideas clearly and concisely. We say that such languages are able to grow.

A language may grow in various ways, new concepts may be supported in libraries using data abstractions, like functions and objects. But in some cases we want to add new syntax which express the new concepts more concisely and naturally than the existing syntax and semantics allows. One mechanism to allow for such growth is syntactic abstraction: It is possible to add new syntactic constructs to the language in libraries, defining new constructs in terms of old ones. In this manner, the language can gracefully adapt to unanticipated needs as they become apparent.

The new syntax and semantics should be indistinguishable from the old one, and subject to the same guarantees as the language provide for the old syntax and semantics. If the host language is a strongly typed language, where well-typed programs are ensured not to go "wrong" in some way, then the system for syntactic abstraction should support the same guarantees for the new code.

We present a syntactic abstraction system for the Fortress programming language based on parsing expression grammars, which allows syntactic extensions of Fortress to be specified in a modular fashion. This allows composition of independent macros, which is important because it enables programs written in domain-specific languages to be embedded in Fortress programs and parsed along with their host programs. The new syntax is indistinguishable from old syntax from the user's perspective which leads to a coherent language, instead of a language with a number of patches. The system is also powerful and sup-

ports case dispatch and allows syntax extensions to expand into other syntax extensions or even it self, thus supporting recursion.

Moreover, the definition of many constructs that are traditionally defined as core language primitives (e.g. for loops) can be moved into Fortress' own libraries, thereby reducing the size of the core language.

The Fortress language is a strongly typed language which guarantees that well-typed programs does not go "wrong". The guarantees should naturally cover the extended syntax as well. We define a core calculus for Fortress which include the system for syntactic abstraction, and formally define a type system for this extended calculus. We also state the theorems which should be proven to ensure the validity of the guarantees.

## An Overview

The dissertation is structured as two major parts, Part I on languages for SMC and Part II on enabling languages to grow over time using syntactic abstraction.

Part I consists of three chapters; in Chapter 1 we give a detailed definition of SMCL and its security guarantees, in Chapter 2, we describe a real world application of an SMCL program, and in Chapter 3 we present our preliminary work on PySMCL, a further development of SMCL and an interesting future direction of research in language support for secure multiparty computation.

In Part II we investigate how syntactic abstraction can enable a language to evolve over time to accommodate the changing needs of its users. It is possible to add new syntactic constructs to the language in libraries, defining new constructs in terms of old ones. In this manner, the language can gracefully adapt to unanticipated needs as they become apparent. Chapter 4 presents our system for syntactic abstraction for the Fortress programming language, and in Chapter 5 we present a core calculus for the system based on the Basic Core Fortress calculus, a type system which guarantees that no type errors occur in expanded macros, and some considerations and future work on the type system.

The work in this dissertation is made up of a combination of previous contributions and novel work. Chapter 1 is similar to the work presented in the paper [71] with minor typographical changes to make it fit the layout of the dissertation. The paper [71] is a further development of the paper [69] and the author's progress report [68], adding descriptions of the dynamic and static semantics of SMCL, formal proof of security guarantees, and a description of optimizations. Chapter 2 is based partly on the paper [16] where we have omitted a formal description of the used cryptographic protocols and added a detailed description of the SMCL program used for the auction. Chapter 3 is joint work with Ivan Damgård, Sigurd Torkel Meldgaard, and Michael I. Schwartzbach. Chapter 4 is similar to the work presented in the paper [8] with minor typographical changes.

We expect the reader to be familiar with static analysis and basic cryptography [91] on a level equivalent to the knowledge taught in most undergraduate courses. Furthermore we expect the reader to have a rudimentary understanding of secure multiparty computation [30] and language-based security [78].

# Part I

# Languages for Secure Multiparty Computation

# Chapter 1

## SMCL - Security Guaranteed

*The limits of language are the limits of one's world*
— Ludwig Wittgenstein

Janus Dam Nielsen  Michael I. Schwartzbach

**Abstract**

We present a domain-specific programming language for Secure Multiparty Computation (SMC).

Information is a resource of vital importance and considerable economic value to individuals, public administration, and private companies. This means that the confidentiality of information is crucial, but at the same time significant value can often be obtained by combining confidential information from various sources. This fundamental conflict between the benefits of confidentiality and the benefits of information sharing may be overcome using the cryptographic method of SMC where computations are performed on secret values and results are only revealed according to specific protocols.

We provide a conceptual analysis of SMC that leads to the design of a domain-specific language, SMCL, for expressing such computations. The linguistic concepts of SMC bridge the gap between high-level security requirements and low-level cryptographic operations constituting an SMC platform, thus improving the efficiency and security of SMC application development. The language is implemented in a prototype compiler that generates Java code exploiting a distributed cryptographic runtime. Various optimizations are provided to speed up the costly computations on secret values.

We provide a formal semantics of SMCL and prove that well-typed programs possess the novel property of *trace security*, which means that they are immune to a wide range of attacks by hostile observers.

### 1.1 Introduction

We present the *Secure Multiparty Computation Language* (SMCL) a domain-specific language for SMC. The development of SMCL is based on an analysis of the SMC domain and experiences with previous applications based on SMC.

The language is carefully designed to provide high-level abstractions and strong security guarantees. The abstractions (including such concepts as *secret*, *public*, and *private* values, *clients*, *servers*, and *tunnels*, and the ability to explicitly *open* secret values) make the application of SMC to many problems like auctions and benchmarking easier to program and also accessible for newcomers to SMC. As a major example, the SMCL language has been used to define the first ever real-life, commercial application of SMC [16].

Security is a central issue in SMC, given that its central premise is that no information at all may be obtained by outside observers (modulo the standard cryptographic assumptions). In principle, SMC will be robust against any kind of passive attack provided that a certain threshold of participants remain uncorrupted. We formalize this idea by defining the notion of *trace security* and we show relatively to a formal semantics that well-typed SMCL programs possess this property.

Another issue is whether information that is opened will unintentionally leak other information. This requires the programmer to keep track of subtle dependencies among computed values. The SMCL language applies a concept of *checked annotations* to support the programmer and to raise awareness of which information is released. An important part of practical SMC programming is to decide which information it is harmless to reveal during computations, since a program with exclusively secret values is generally prohibitively expensive to execute.

Finally, the SMC runtime is very sensitive to the use of secret values in computations, particularly multiplications, which due to the involved cryptographic protocols are several orders of magnitude more expensive than additions. Consequently, we apply analysis that seek to reduce the number of such operations as much as possible.

### 1.1.1 Contributions

The goal of this article is to give a firm and detailed description of all aspects of the SMCL language. Our contributions in this article are:

1. A conceptual analysis of the domain of SMC programming.
2. A design and implementation of a domain-specific language for secure multiparty computation (SMCL)
3. A concurrent semantics for SMCL
4. An effect-based type system for SMCL which approximate our concept of hoistability and provides better precision than previous type systems for noninterference, and a proof of its soundness.
5. A characterization of a very broad family of attacks which can be performed by exploiting physical measurements e.g. timing attacks.
6. We define the concept of *trace security* for SMCL programs which is based on a formalization of physical measurements and we show that SMCL

programs are trace-secure under the reasonable assumption that all the operations of the runtime we use are trace-secure.

7. We present an optimization for multiplications and prove that it is correct.

8. A simple language of checked annotations for describing potential information dependencies among variables in SMCL programs.

Of these, numbers three to seven are novel contributions compared to the work published in [69].

### 1.1.2 Structure

To achieve the goal of this article we first give an analysis of secure multiparty computation as a domain and present the key linguistic concepts we have identified, Section 1.2. Then we describe the language in Section 1.3 and the formal semantics in Section 1.4. In Section 1.5 and Section 1.6 we discuss security and prove that SMCL programs are secure against trace-attacks. We present an optimization of multiplications in Section 1.8. A description of related work is given in Section 1.9, and we conclude the article in Section 1.10 and give directions for future work.

## 1.2 Secure Multiparty Computation as Domain

We give a short introduction to SMC and SMCR a runtime for SMC. Based on this we provide a conceptual analysis to determine the central concepts of SMC.

### 1.2.1 SMC

The seminal example of SMC is the *Millionaire's problem* which involves a number of millionaires who want to find out which is richer, but all of them refuse to disclose their net worth. A conventional solution would involve an external trusted third party (TTP) that could perform the comparisons and report the result. An external TTP is usually realized by a law or accountant company, which is paid an adequate sum as incentive to not reveal the secret information. The amount paid must be high enough that a counter-bribe from any of the millionaires is unlikely, however no-matter how high a sum is paid it cannot be guarantied that the TTP will not reveal some information intentionally or unintentionally. Thus this approach to establishing a TTP is expensive and without any guarantees of security. SMC offers another approach to realizing a TTP using software which is secure and cost efficient. Yao [109] showed in 1982 that it is possible to find the richest of two millionaires without any degree of trust between the two parties if they work together.

In general a secure multiparty computation involves a number of parties. Each party does not trust the other parties, but they still want to collaborate in performing a computation. The computation could be in the form of a *Secure Function Evaluation* (SFE), where we have $n$ parties $P_1, \ldots, P_n$ that wish to

jointly compute the value of an integer function $f(x_1, x_2, \ldots, x_n)$, where party $P_i$ only knows the input value $x_i$, and the input must be kept secret from the other parties. SMC can also be performed in an *Iterative SFE model* which generalizes SFE. Here the parties perform the following sequence of actions any number of times:

- A subset of parties input values
- All parties work together and perform SFE
- A subset of values are revealed to some subset of parties

The Millionaires examples fits nicely in the SFE model, whereas the card game of poker fits better in the Iterative model.

The foundation for SMC that we have made use of in our work is *threshold secret sharing scheme* [86] defined by a probabilistic algorithm $\mathcal{S}$. It takes as input a secret $s$ chosen from some finite set $S$, and outputs a sequence of $n$ shares $s_1, \ldots, s_n$. A share is a sequence of bits. Furthermore the scheme comes with a threshold $1 \leq \tau \leq n$, and the properties:

**Privacy** Take any subset $I$ of the indices $\{1, \ldots, n\}$ and run $\mathcal{S}$ on any secret $s$. Then the probability distribution of the $\{s_i \mid i \in I\}$ is independent of $s$.

**Correctness** Take any subset $J$ of the indices $\{1, \ldots, n\}$ of size at least $t + 1$, and run $\mathcal{S}$ on any secret $s$. Then $s$ is uniquely determined by $\{s_i \mid i \in J\}$, and there is an efficient algorithm that computes $s$ from $\{s_i \mid i \in J\}$.

These properties ensure that if someone knows up to $t$ shares of a secret $s$ then the secret is not compromised, however if someone knows $t + 1$ or more shares then he can efficiently reconstruct the secret. Understanding this model is key to understanding a number of our decisions.

### 1.2.2 SMCR: A Runtime Environment for SMC

SMCR is an implementation and API for secure multiparty computation, and is a further development of the system used in [17]. SMCR enables secure multiparty computations to take place by allowing each party to make their input values secret, exchange them, perform arithmetic operations jointly on such values, and reveal values only by collaboration from all parties.

Under standard cryptographic assumptions it can be proved that no party can obtain any extra information. SMCR is robust in the sense that the secrecy can only be compromised if a certain fraction of the parties decide to collude in *passive corruption*, where they pool all their secret values but continue to follow the protocol. The standard threshold is $t = n/2 + 1$ parties, but (more expensive) protocols exist where the threshold is $t = n - 1$, i.e. where each party trusts no other. SMCR is currently not robust against *active corruption* where the parties choose to sabotage the computation by not adhering to their

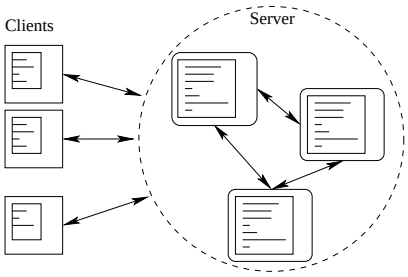



Figure 1.1: Conceptual and concrete view of clients and server

individual part of the protocol, but such behavior is guaranteed to be detected and some (even more expensive) protocols can even tolerate a threshold of $t = n/3$ such parties.

SMC computations will generally involve complex protocols that involve many rounds of communication between all parties [17]. Thus, simple operations become several orders of magnitude more expensive than their non-cryptographic counterparts. Technically, the SMCR runtime is a Java API with support for public key encryption, secret sharing, primitive SMC operations, and distributed deployment and communication. We will not discuss the cryptographic challenges and technicalities in realizing the SMCR any further, but refer the reader to [17].

### 1.2.3 Conceptual Analysis

Based on experiences with current and earlier versions of SMCR [18, 51, 96], we can identify a number of concepts that are used in describing realistic SMC computations.

First, a practical application will typically involve a number of *clients* that provide the inputs and receive some computed results. The computation itself is performed by a *server* which is conceptually a single machine that is realized through a number of separate parties that perform the SMC computations by running identical copies of the code in lock-step, see Figure 1.1. In a realistic example, involving the Danish commodities market for sugar beets, there are around 1200 farmers as clients and the server is implemented by parties representing the buyers, the sellers, and a Government office. The example is described in more detail in Chapter 2. In general, there will be (possibly overlapping) one-to-many mappings from the various kinds of clients and the single server to physical machines.

Note that the clients are in principle unrelated to the parties mentioned in Section 1.2.1, as every secret client input is represented as a share of input on each of the server parties. Also, from the programmer's point of view, the server is a single entity.

Clients communicate with the server only and have no incentive to communicate directly with each other, since they generally do not trust each other. Potential attacks based on clients communicating directly and not through the server are captured in the adversary's capabilities as described in Section 1.6.

The division into clients and a single server separates public computations from secure computations respectively, in the sense that SMC computations are performed only on the server. Note that we actually have three kinds of values (and corresponding computations):

- *secret* values that reside on the server and are owned jointly by the server parties;
- *public* values that reside in plain view on the server; and
- *private* values that reside only on a single client.

Public and private computations are performed on ordinary values with a standard runtime representation. A secret value has a different runtime representation consisting of secret shares (generated using a threshold secret sharing scheme) residing on the machines that physically realize the server parties, and the execution of primitive operations on such values will typically involve complex protocols with several rounds of communications. The server will have the ability to explicitly *open* a secret value which requires collaboration from all server parties. Careful limitations must be placed on the use of secret values as conditions in the control flow to avoid attacks that observe public side-effects of computations.

Clients and the server require secure and flexible communications: In some scenarios, a client only submits an input and does not need to wait for the result, whereas in other scenarios the interaction is more complex and ultimately requires a client to be connected to the server for the duration of the computation. For these purposes we identify the need for *tunnels* for asynchronous communication and *remote procedure calls* for synchronous communication.

SMC applications are generally instances of Iterative Secure Function Evaluation. To support Iterative SFE we allow the server to perform computations on public values and to perform iterations on public values. In SFE iteration and branching (in the usual sense) on secret values cannot occur, so SFE is similar to evaluation of a straight-line program. As a motivating example we may consider the use of second-level protocols where a server repeatedly performs a sequence of secure auctions until some market equilibrium has been attained. Conceptually, the server will execute Turing-complete programs in which the data is separated into *public* and *secret* types. However, computations that involve only secret values still only corresponds to loop-free programs.

While the underlying cryptographic protocols are known to be provably secure, it is still a challenge to write reliable SMC applications, since confidential information may be propagated along non-obvious paths and may be leaked in subtle ways, thus an unbounded number of potential attacks exists. Implicit flows and timing attacks are classical examples [36, 56]. As another example, consider a variable $x$ containing a secret integer value. Revealing the values of $x\%10$ and $x/10$ is sufficient to effectively reveal $x$ itself. Thus, programmers must keep track of such value flow dependencies, which turns out to be a tedious task. However, since any non-trivial application is bound to reveal *something* about its input, the programmer must use careful judgment to determine what is

acceptable. Thus, we are looking for a concept of *checked annotations* ensuring that a programmer has been made aware of all potential information leaks and has explicitly considered them.

To conclude, we have identified the following key concepts within the area of SMC programming:

- *Architecture*: The client-server view forms the fundamental computing paradigm of SMC, providing a separation between private, public, and secret computations and between logical and physical parties.

- *Values*: Values are either secret, private, or public, which also determines their runtime representation and separates the efficiency of primitive operations by several orders of magnitude.

- *Communication*: Clients communicate with the server only, either by using tunnels or by reacting to remote procedure calls from the server.

- *Expressiveness*: A general SMC framework must be able to perform any computation; i.e., it must be Turing-complete on private and public values.

- *Security*: Writing reliable SMC programs that do not leak unintended information, is a tedious and error-prone task that can benefit from automated assistance.

## 1.3 The SMCL Language

Based on the key concepts identified in Section 1.2 we have designed a novel language called the *Secure Multiparty Computation Language* (SMCL). It is a high-level, domain-specific language [99], which allows programmers to express concepts such as clients, server, and operations on secret values directly using a special syntax and control structures tailored to the domain of SMC.

SMCL enjoys the classical advantages of being a domain-specific language as opposed to being a library API for a general-purpose language:

- The specialized syntax of SMCL closely matches the problem domain.

- A domain-specific compiler may generate more efficient code for SMCL.

- It is possible to perform domain-specific analyses that consider global properties of SMCL programs and provide stronger safety guarantees.

We start by introducing an example which gives a quick introduction to the look and feel of SMCL and then proceed to a more in-depth presentation of the basic concepts and continue with an elaboration of the example. Before concluding we shortly discuss the implementation of SMCL.

```
C1: client Millionaires :
C2:   tunnel of sint netWorth;
C3:
C4:   function void main(int[] args) {
C5:     ask();
C6:   }
C7:
C7:   function void ask() {
C8:     netWorth.put(readInt());
C9:   }
C10:
C11:  function void tell(bool b) {
C12:    if (b) {
C13:      display(``You are the richest!'');
C14:    }
C15:    else {
C16:      display(``Make more money!'');
C17:    }
C18:  }
 S1: server Max :
 S2:   group of Millionaires mills;
 S3:
 S4:   function void main(int[] args) {
 S5:     sint max = 0;
 S6:     sclient rich;
 S7:     for (client c in mills) {
 S8:       sint netWorth =
                      c.netWorth.get();
 S9:       if (netWorth > max) {
S10:         max = netWorth;
S11:         rich = c;
S12:       }
S13:     }
S14:     for (client c in mills) {
S15:       c.tell(open(c==rich|rich));
S16:     }
S17:  }
```

Figure 1.2: The generalized Millionarie's problem in SMCL

### 1.3.1 An Example

We present an example program written in SMCL in Figure 1.2, which shows an implementation of the solution to the Millionaire's problem, generalized to an arbitrary number of millionaires.
The *Millionaires* client describes the actions of a millionaire and the *Max* server calculates and reports who is the richest. Each *Millionaires* client has a *main*

function that initiates its execution. The other functions may either be invoked by the client itself (as in line C5) or by the server as a remote procedure call (as in line S15). In the example, each client submits its net worth via the *netWorth* tunnel (line C8). Tunnels are described in more detail in Section 1.3.7.

The server declares that *Millionaries* may belong to a group named *mills* (line S2). The group is processed in the *main* function of the server which describes the SMC application that is executed jointly by all the server parties. The *Max* server uses two secret variables *max* and *rich* to retain the current highest net worth and the identity of the corresponding millionaire (lines S5 and S6). It then proceeds by using a `for` iterator to process each client in turn (line S7), updating *max* and *rich* if required. The update is guarded by the secret condition of the `if` command (line S9). To finish, the server reports to each client a boolean indicating whether or not that client is the richest. The *open* operator downgrades a value from secret to public.

### 1.3.2 Basic Concepts

We now present the basic concepts of SMCL and the role they fulfill within SMCL, how they are declared, and what restrictions are put on their use.

### 1.3.3 Clients and the Server

There is a clear distinction between the role of the server and the clients. The server does the computation and has no input or output besides communication with the clients. Clients take the input from the user, process it, provide it to the server, receive output from the server, and display it to the user. This can go on any number of times.

The server and client concept is central in the SMC world, it allows a wide range of scenarios from the paranoid self-trust scenario where the participants do not trust each other, to the more liberal scenario where some participants trust a specific party to do their part of the computation. Clients are declared using the reserved word `client` followed by the name of the client and the client body. Similarly the server is declared using the reserved word `server` followed by the server body. The client and server bodies are declared in the same way, as a number of functions interleaved with a number of field declarations.

### 1.3.4 Values

Values are separated into public and secret values. Secret values are secret booleans, secret clients, and secret integers. Private values are booleans, integers, and records. The public values are the same as the private values plus clients.

### 1.3.5 Functions and Control

#### Functions

The server and clients may declare any number of functions, however at least one function, the void *main*(int[] *args*) function must always be present. Functions may return any kind of values and may take any kind and any number of arguments. Functions are declared using the reserved word `function`, prefixed by a return type and followed by a comma-separated list of argument declarations, and finally a block command. An argument declaration is a type followed by a variable, the variable has scope in the entire block command, but may be shadowed by other variable declarations. Functions may be recursively defined.

Functions declared in a client may be invoked from the server, the actual execution of the function will be carried out on the client similarly to a remote procedure call.

If the function has a secret return type and is invoked from the server then the return value transmitted back to the server is encrypted, and split into secret shares for the server parties. If the function is invoked from within the client and the return type is secret then it is treated as if the return type is public.

In addition to user defined functions there are two primitive functions for input/output. The *readInt* and *display* functions are rudimentary primitives for communicating with the person controlling the client (in a future version, this will happen through a browser with support for appropriate GUI primitives). A third primitive function, $open\,(e|x, y, z)$ is provided for declassification of secret values to public values. The operator computes and opens the secret expression $e$ and declares that the programmer recognizes the simultaneous indirect leaking of some information about the secret variables $x$, $y$, and $z$. A program cannot be compiled unless it is *well-annotated*, meaning that the programmer has recognized all potential leaks (see Section 1.6.5 for further details).

**Restrictions** A number of restrictions are needed to enforce security properties. We mention the restrictions here for completeness of the description and give the motivation behind the restrictions in Section 1.6.

No recursion with secret stop condition. No calls to functions with side-effects within conditionals guarded by secret values. Clients may only invoke functions defined within the client itself.

### If-commands

An if-command (or simply a conditional) is constructed from the reserved word `if`, a conditional expression followed by a command and an optional occurrence of the reserved word `else` along with another command, for example,

```
 S9: if (netWorth > max) {
S10:     max = netWorth;
S11:     rich = c;
S12: }
```

**Restrictions** Conditionals guarded by a expression of secret type must be treated specially and we *dissallow* all of the following in the branches of such a conditional:

- assignments to public non-locally declared variables
- invocation of functions which have side-effects, including I/O
- occurences of `return` commands
- occurences of `while` loops

But we do allow assignment to public locally declared variabels.

### Variable declaration

A variable is declared by first declaring its type and then its name. The name can be any sequence of characters, but must begin with a letter. Variables may be declared by the server or client, as a field which is in scope throughout the server or client, but may be shadowed. Variables may also be declared in block-commands, with scope throughout the block, but may be shadowed.

### While-commands

A while-command consists of the reserved word `while` followed by an expression (the condition) and a block command, for example,

```
int  i = 7;
sint  x = 84;
while (i < 42) {
  x = x + i;
  i = i + 1;
}
```

**Restrictions** `while` loops may not have a condition with a secret type.

### For-commands

There are two variations of the for-command. One is used for performing the same computation a statically known number of times, and the other is used for iterating through a group of clients.

The first for-command is written with the reserved word `for` and then a variable declaration, followed by a lower and upper bound, which should be numbers. The declared variable is in scope inside the block-command and draws a new value from the interval including the lower and excludes the upper bound. An example is:

```
S14: for (int inx ; 0 ; 10) {
S15:     a[inx] = 42;
S16: }
```

Where we initialize the array $a$ with the value 42 for the indicies 0 to 9.

The second for-command is written with the reserved word `for` and then a variable declaration, followed by the reserved word `in`, followed by an expression which evaluates to a group, and finally a block-command. The declared variable is in scope inside the block-command and draws a new value from the group in each iteration of the for-command. An example is:

```
S14: for (client c in mills) {
S15:     c.tell(open(c==rich|rich));
S16: }
```

**Restrictions** Only groups can be iterated using for-commands.

## 1.3.6 Types

The SMCL language supports the primitive datatypes int and bool. The identities of clients also form a datatype client. All of these have secret versions, denoted sint, sbool, and sclient. The types sbool and sclient are represented as secret integers at runtime, because the SMCR only manipulates public values and secret integers. A secret client is a client whose identity (IP-address) is secret shared; the total number of clients is always public. Furthermore, it is possible to construct records and multi-dimensional arrays of such primitive datatypes. Private, public, and secret datatypes support the same standard primitive operations, and the type system ensures that results are secret unless all arguments are public (this may involve implicit conversions to the runtime representation of secret values).

### 1.3.7 Tunnels

A tunnel is a mean for asynchronous communicating between a client and the server, and is declared in clients only. A tunnel is declared using the reserved words `tunnel` and `of`, followed by the type of values that can be placed in the tunnel, and terminated by the name of the tunnel followed by a semicolon, for example:

C2: `tunnel of` sint *netWorth*;

A tunnel may be declared to contain data of any primitive type, int, sint, bool, sbool. When a client sends a secret value to the server, the transmitted value is not only encrypted, but it is also split into secret shares for the server parties, matching the runtime representation of secret values. When the server sends a secret value to a client, all server parties send encrypted version of the secret shares which are then assembled on the client to yield a private value. Public values sent are encrypted but not secret shared. A tunnel is equipped with functions for sending *put* and retrieving *get* data and for inquiring the emptiness *isEmpty* of the tunnel.

Currently a tunnel is only working as long as the client and the server are running. That is if one of them terminates, any information in the tunnel is lost and the remaining process is likely to terminate abruptly. In a future version we would like to introduce various different kinds of persistent tunnels, e.g. an XML-tunnel, database-tunnel, or a plain file-tunnel, where all values are serialized and stored persistently. The values can be store in different places. In a central computer if all values are encrypted using the public keys of the server parties. Another possibility is to store the values for each party at a computer owned by the organization running the server party.

**Restrictions** Only primitive types can be sent through a tunnel.

### 1.3.8 Groups

A group is a collection of clients. Currently a group may only contain clients of the same kind, they have to be homogeneous. Groups may only be declared in the server. A group is declared using the reserved words `group` and `of` followed by the name of the clients and the name of the group, for example,

S2: `group of` Millionaires *mills*;

The members of a group are specified externally by a mapping supplied to the SMCR runtime describing the concrete participants involved during runtime. Figure 1.3 shows a hypothetical example. Each participant is identified by an IP address, a port number, and a public encryption key. Note that the same machine may serve both as a client and as a server party. Clients may further be listed as belonging to a number of groups, in this case only the single group *mills* containing all clients.

**Restrictions** Only homogeneous groups.

### 1.3.9 The Example Elaborated

In generalizing the original Millionarie's problem from two to many millionaires, we have in our solution chosen that while the net worth of each millionaire remains secret, it is actually public information which millionaire is the richest, see Figure 1.4(A). In a stricter version of the generalized problem we could also keep this information secret and only allow each millionaire to know his own status. In our program, we would then change lines S14 through S16 into the lines of Figure 1.4(B).

```
client: Millionaires
 gates.microsoft.com    4001  0x85FFA494  mills
 ebenezer.scrooge.org   4001  0x5532BB72  mills
 ingvar.ikea.com        4001  0x2333DDCC  mills
 larry.google.com       4001  0x631DE7F2  mills
 sergei.google.com      4001  0x7587B5AF  mills

server
 gates.microsoft.com    4000  0x857722B7
 smcl.brics.dk          4000  0xF471BCA7
 survey.fortune.com     4000  0x66A7FF35
```

Figure 1.3: A map identifying the concrete participants

```
1: for (client c in mills) {
2:     c.tell(open(c == rich|rich));
3: }
```

(A) public booleans, public receivers

```
1: for (client c in mills) {
2:     c.tell(c == rich);
3: }
```

(B) secret booleans, public receivers

```
1: for (sclient c in mills) {
2:     c.tell(open(c == rich|c, rich));
3: }
```

(C) public booleans, secret receivers

```
1: for (sclient c in mills) {
2:     c.tell(c == rich);
3: }
```

(D) secret booleans, secret receivers

Figure 1.4: The combinations of server knowledge

Here, we do not open the secret boolean before it is sent to the client. This means that the server parties send their shares representing the value of type sbool to the client which combine the shares into a value of type bool. An equivalent effect can be achieved by changing the iterator $c$ to have type sclient and thus keep it secret while revealing the comparison result, Figure 1.4(C). Consequently, the invocation $c.tell(\ldots)$ is now implemented by sending to all clients the same message that can only be understood by the intended recipient (function invocations with illegible arguments are ignored by the clients). Since $c$ is now also secret, the *open* operation must also recognize responsibility for compromising it (ever so slightly). In a yet stricter version, we may change the three lines into the lines of Figure 1.4(D).

For this particular example, however, this refinement makes no difference (since the server always sends one *true* value and a number of *false* values).

### 1.3.10 Implementation

SMCL is implemented by a prototype compiler (SMCLc) which produces Java code based on the SMCR API. A Java program is created for each kind of client and for the server. Deployment scripts can be used to install and start applications. Currently, all communication takes place through a coordinator process (that only sees encrypted information). The coordinator could itself be distributed using broadcast protocols.

## 1.4 Semantics

In this section we formally describe the non-standard parts of the SMCL semantics. We will be needing a good deal of terminology which we introduce as needed, first we introduce the abstract syntax of SMCL programs and then the formal semantics. In Figure 1.5 we see the overall structure of an SMCL program which defines a number of different clients $K$ and a server $\Sigma$. We have omitted the abstract syntax for *client definitions* because computations in clients are mostly standard for a strongly typed imperative language, and most importantly does not involve secret values; we refer to [70] for a full description of the semantics. In Figure 1.6 we show the abstract syntax for the *server definition*, here we have typeset the various parts of the abstract syntax tree (AST) using different fonts: *nonterminals*, `keywords`, and *variables*. The syntactic constructs are either standard or easily identified from the discussion in the previous section. We use the following naming conventions: Field declarations are denoted $D$, function names $f$, function declarations $F$, commands $C$, expressions $e$, types $\tau$, and tunnels $\theta$, see Section 1.5 for the type system. Values are separated into public and secret values. Secret values, secret booleans, secret clients, and secret integers, are denoted by $\boxed{v}$ (intuitively, the value is hidden inside a box, so we cannot see the value). Public values are denoted by $v$. A value which is either public or secret is denoted by $w$, we denote the set of all secret, public, and private values as *Value*.

$$\textit{Program} \quad ::= \quad K^+ \ \Sigma$$

Figure 1.5: SMCL abstract syntax and values

SMCL is a concurrent imperative language, concurrency means that the server and clients may be executing at the same time. The server and clients may be executed with an arbitrary interleaving of commands and expressions, and only need to synchronize for communication. An SMCL server definition consists of a number of functions containing commands, which are executed sequentially. Commands may contain expressions which potentially have side effect on the state of the program.

An SMCL program consists of a server $\sigma$ and a set of clients $\kappa_i \in \chi$. We use $\xi$ to range over both the server and clients. The evaluation of an SMCL program is defined in terms of a small-step operational semantics [74]. We define the semantics using a number of transition systems. A transition system consists of a set of configurations $\gamma_\alpha \in \Gamma_\alpha$, a set of terminal configurations $\mathrm{T}_\alpha \subseteq \Gamma_\alpha$, and a transition relation $\Rightarrow \subseteq \ \Gamma_\alpha \ \times \ \Gamma_\alpha$, where $\alpha$ is the name of the transition system.

We define different transition systems to model the difference between evaluation of commands and expressions, and also to model the difference between evaluation on the server and clients. We use transition systems for client-side commands (the cc system), client-side expressions, server-side commands (the sc system), server-side expressions, and also one for specifying the interaction and communication between the server and the clients (the Server-Client scl system). The transition systems are fairly standard and we will only discuss the non-standard parts below to some extent (we intentionally left out the names of the transition systems that we will not discuss in this article). We refer to [70] for all the details.

The set of configurations $\Gamma_{\mathrm{scl}}$ and terminal configurations $\mathrm{T}_{\mathrm{scl}}$ for the scl transition system is defined as follows:

$$\begin{aligned}
\Gamma_{\mathrm{scl}} &= \Gamma^1_{\mathrm{cc}} \times \ldots \times \Gamma^n_{\mathrm{cc}} \times \Gamma_{\mathrm{sc}} \times \mathit{State}_{sv} \\
&\cup \ \Gamma^1_{\mathrm{cc}} \times \ldots \times \Gamma^n_{\mathrm{cc}} \times \mathit{State}_{sv} \\
\mathrm{T}_{\mathrm{scl}} &= \mathrm{T}^1_{\mathrm{cc}} \times \ldots \times \mathrm{T}^n_{\mathrm{cc}} \times \mathit{State}_{sv}
\end{aligned}$$

| | | | |
|---|---|---|---|
| $\Sigma$ | ::= | `server` *name* : $D^*$ $F^+$ | server definition |
| $F$ | ::= | `function` $\tau$ $f$ $(\tau_1\ x_1,\ldots,\tau_n\ x_n)$ { $C$ } | function |
| $D$ | ::= | $\tau\ x = e$ | field definition |
| | \| | `group of` *name* $x$ | group definition |
| $C$ | ::= | $\tau$ *id* | identity |
| | \| | $\tau\ x = e$ | assignement |
| | \| | `while` ( $e$ ) { $C$ } | while-loop |
| | \| | `for` ( $\tau\ x$: $e$ ) { $C$ } | for loop |
| | \| | `if` $e$ { $C_1$ } `else` { $C_2$ } | if statement |
| | \| | $C_1;C_2$ | sequence |
| | \| | `return` $e$ | return expression |
| | \| | `return` | return |
| | \| | `void` | void |
| $e$ | ::= | $w$ | |
| | \| | $x$ | variable |
| | \| | $f(e_1,\ldots,e_n)$ | function application |
| | \| | $open(e\mid\ x_1,\ldots,x_n)$ | open operation |
| | \| | $\theta.put(e)$ | put on tunnel |
| | \| | $\theta.get(e)$ | get from tunnel |
| | \| | $\theta.take(e)$ | take from tunnel |
| | \| | $e_1\ op\ e_2$ | operations |
| *op* | ::= | + \| - \| * \| / \| | |
| | | == \| && \| \|\| \|=\|>\|>=\|<=\|< | |

Figure 1.6: Server-side SMCL abstract syntax and values

A configuration in the scl transition system consist of a number of configurations from the respective transition systems, one for each client process $\Gamma^i_{\mathrm{cc}}$ and one for the server $\Gamma_{\mathrm{sc}}$. We write configurations, $G \vdash \|_{\kappa_i \in \chi} \langle \kappa_i : C \rangle \,\|\, \langle \sigma : C, \sigma.S \rangle$, as a number of concurrent configurations for the clients in the set $\chi$, similar to [98], and an additional configuration for the server. We write to $\kappa_i : C$ to mean that client $\kappa_i$ is about to evaluate the command $C$ and similarly for the server $\sigma$.

We define the *Global client store* $G$ in Figure 1.7 to be a tuple containing the client stores $\kappa_i.S$ and all the tunnels $\kappa_i.\theta_j$. We write $G[O \mapsto U]$ for the store $G$ where the component $O$ is updated to hold $U$. The server and each client $\xi$ have a local store $\xi.S$ defined as a map from variables to values: $\xi.S = Var \mapsto Value$. The server store $\sigma.S \in State_{sv}$ is a member of the set of all server stores $State_{sv}$. Similarly for the client store, $\kappa.S \in State_{cl}$. The local stores also work as environments in the way one would expect. A *preliminary* store is the store used when computation begins. A preliminary local store is initialized with bindings from function names to function bodies and bindings from field variables to an initial value (which depends on the type of the field). From here on fields are treated as ordinary variables which may be shadowed in the current scope.

The transition rules of the scl system are described in Figure 1.8 where the first rule describe how clients are evaluated and the second rule describes how the server is evaluated. In each case a client or the server is chosen non-deterministically and may proceed by evaluating one step using the appropriate transition system.

The other transitions systems will not be presented here in their entirety as men-

$$G \quad : \quad [\overline{\kappa.S}, \overline{\kappa_1.\theta}, \ldots, \overline{\kappa_n.\theta}]$$

Figure 1.7: The global client store of an SMCL computation

tioned above. However, we will present the standard rule for server-side assignment as an introduction to the sc system, and then we describe the nonstandard evaluation of conditional commands on secret values. As we will discuss in Section 1.6 it is very important to prevent physically observable side-effects of computations, e.g. timing attacks [5, 43, 56]. A part of our solution to avoid physical side-effects is to execute both branches of a conditional command and combine the results in the end according to the value of the test. Why this solution works, and what implications it has will be thoroughly discussed and correctness of the solution will be formally established in Section 1.6.

The server-side commands system is defined as follows:

$$\begin{array}{rcl}
\Gamma_{\text{sc}} & = & \text{sc} \times \mathit{State}_{sv} \\
& \cup & \text{sc} \times \mathit{State}_{sv} \times \mathit{State}_{sv} \\
& \cup & \text{sc} \times \mathit{State}_{sv} \times \mathit{State}_{sv} \times \mathit{State}_{sv} \\
& \cup & \texttt{void} \times \mathit{State}_{sv} \\
& \cup & (\mathsf{Rtn}\ w \cup \mathsf{Rtn}\ ) \times \mathit{State}_{sv} \\
\mathrm{T}_{\text{sc}} & = & \texttt{void} \times \mathit{State}_{sv} \\
& \cup & (\mathsf{Rtn}\ w \cup \mathsf{Rtn}\ ) \times \mathit{State}_{sv}
\end{array}$$

The configurations are divided into a number of intermediate configurations and terminal configurations. The intermediate configurations contains two additional configurations defined and used for evaluation of conditional commands on secret values. The difference is the addition of one or two additional server stores respectively. The terminal configurations correspond to normal termination $\langle \texttt{void}, \xi.S \rangle$ of a command and abnormal termination (return of a function call) $\langle \mathsf{Rtn}\ w, \xi.S \rangle$ which carries the return value $w$. Return values from functions are modeled in the configuration using the $\mathsf{Rtn}$ component in order to allow values to flow from `return` commands to call-sites.

The rules for evaluating server-side assignments are shown in Figure 1.9. The configuration $G \vdash \langle y = e, \sigma.S \rangle$ in rule Assign-eval consists of the term $x = e$, the server store $\sigma.S$ and the global client store $G$. The term is an assignment of the result of the *argument expression* $e$ to the identifier $y$. In rule Assign-eval we evaluate the argument expression one step and obtain the resulting argument expression $e'$ and server store

$$\frac{G \vdash \langle \kappa_i : C \rangle \ \rightarrow_{\text{cc}} \ G' \vdash \left\langle \kappa_i : C' \right\rangle}{G \vdash \|_{\kappa_i \in \chi} \langle \kappa_i : C \rangle \,\|\, \langle \sigma : C, \sigma.S \rangle \ \rightarrow_{\text{scl}} \ G' \vdash \|_{\kappa_i \in \chi} \left\langle \kappa_i : C' \right\rangle \,\|\, \langle \sigma : C, \sigma.S \rangle} \ (\textsc{client-eval})$$

$$\frac{G \vdash \langle \sigma : C, \sigma.S \rangle \ \rightarrow_{\text{sc}} \ G \vdash \left\langle \sigma : C', \sigma.S' \right\rangle}{G \vdash \|_{\kappa_i \in \chi} \langle \kappa_i : C \rangle \,\|\, \langle \sigma : C, \sigma.S \rangle \ \rightarrow_{\text{scl}} \ G \vdash \|_{\kappa_i \in \chi} \langle \kappa_i : C \rangle \,\|\, \left\langle \sigma : C', \sigma.S' \right\rangle} \ (\textsc{server-eval})$$

Figure 1.8: One-step evaluation of an SMCL computation

$$\frac{G \vdash \langle e, \sigma.S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle e', \sigma.S'\rangle}{G \vdash \langle y = e, \sigma.S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle y = e', \sigma.S'\rangle} \quad \text{(Assign-eval)}$$

$$\frac{\sigma.S' = \sigma.S[y \mapsto w]}{G \vdash \langle y = w, \sigma.S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle w, \sigma.S'\rangle} \quad \text{(Assign-exp)}$$

Figure 1.9: Server-side semantics for assignments

$\sigma.S'$ which is used in the resulting configuration $G \vdash \langle y = e', \sigma.S'\rangle$ along with the unchanged global client store. The global client store is unchanged since there is no interaction with the clients in the evaluation step.

In rule Assign-exp the argument expression has been evaluated to a value $w$ which is supposed to be assigned to the identifier $y$. The assignment simply happens by updating the binding of the identifier $y$ to point to the value $w$ in the server store, regardless of the value being public or secret.

Evaluation of conditional commands on secret values is performed by evaluating both branches of the conditional in sequence. To evaluate the branches we make copies of the server store. In total we need three stores, one for each branch and a copy of the original for combining the effects. The threading of environments complicates the rules, and we need eight rules in total to specify the evaluation. Figure 1.10 presents the entry rule (If-stm) and the three main rules. The other four rules are mainly concerned with evaluation of each branch in a conditional command to obtain the forms shown in Figure 1.10 and will not be presented here.

A conditional command is in general evaluated by first evaluating the condition If-stm, like in the usual semantics for conditionals. If the result of evaluating the condition turns out to be a public value the evaluation proceeds as in the usual semantics. However if the value is a secret value, then both branches are evaluated in a copy of the current store, as indicated by the three stores in the configuration. We evaluate the branches by first evaluating the then-branch followed by the else-branch.

The rule If-sbool-then-final is the last step of evaluating the then-branch $C_1$ to the terminal configuration containing `void` and the resulting store. Return values cannot occur since it is an error to return from a branch of a secret conditional. At this point we duplicate the original store and the following configurations now have three stores.

The execution continues at If-sbool-else where $C_2$ is evaluated in the newly created copy of the original store $U_{else}$. The evaluation results in a store $U'_{else}$ which we save as part of the new state of the conditional. The evaluation of $C_2$ now proceeds in a number of steps using the store $U_{else}$, and eventually the system ends up in a configuration as the one in rule If-sbool-phi. The server state $\sigma.S$ can be stratified by distinguishing between the public and the secret state. We write $(\overline{x}, \overline{s})$ for an unnamed configuration $\gamma_\alpha$ with public state $\overline{x}$ and secret state $\overline{s}$.

The last step of evaluating a conditional command is the combination of stores from each branch. The result is a store $\sigma.S'$ which is the same as the original store $S$ but updated for each variable which has been assigned to during the execution of a branch. We write $s\{[y_1 \mapsto w], \ldots\}$ for updating the store $s$ with the bindings in the set $\{[y_1 \mapsto w], \ldots\}$. We update a variable $x$ by looking up $x$ in both $U_{then}$ and $U_{else}$ and combine the values using conditional assignment. If a variable has not been updated in a branch its value is unchanged compared to the original store and conditional assignment ensures that the result is correct. If a variable in the original environment has been updated with a public value in either branch or with different public values

$$\frac{G \vdash \langle e, S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle e', S'\rangle}{G \vdash \langle \texttt{if}\ (e)\ \{C_1\}\ \texttt{else}\ \{C_2\}, S\rangle \ \rightarrow_{\text{sc}} \ G \vdash \langle \texttt{if}\ (e')\ \{C_1\}\ \texttt{else}\ \{C_2\}, S'\rangle}$$

(If-stm)

$$\frac{G \vdash \langle C_1, U_{then}\rangle \ \rightarrow_{\text{sc}} \ G \vdash \langle \texttt{void}, U'_{then}\rangle}{\begin{array}{c} G \vdash \langle \texttt{if}\ (\boxed{v})\ \{C_1\}\ \texttt{else}\ \{C_2\}, U_{then}, S\rangle \\ \rightarrow_{\text{sc}} \\ G \vdash \langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{C_2\}, U'_{then}, S, S\rangle \end{array}}$$

(If-sbool-then-final)

$$\frac{G \vdash \langle C_2, U_{else}\rangle \ \rightarrow_{\text{sc}} \ G \vdash \langle C'_2, U'_{else}\rangle}{\begin{array}{c} G \vdash \langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{C_2\}, U_{then}, U_{else}, S\rangle \\ \rightarrow_{\text{sc}} \\ G \vdash \langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{C'_2\}, U_{then}, U'_{else} S\rangle \end{array}}$$

(If-sbool-else)

$$\frac{\begin{array}{c} \overline{s}_1 = \overline{s}\{\ [y \mapsto \boxed{v} * \overline{s}_{then}(y) + (1 - \boxed{v}) * \overline{s}_{else}(y)]\ \mid \\ \forall y \in \overline{s} : \overline{s}_{then}(y) = \boxed{v'}\ \wedge\ \overline{s}_{else}(y) = \boxed{v''}\ \} \\ \overline{s}_2 = \overline{s}_1\{[y \mapsto \boxed{v}]\ \mid \forall y : (v = \overline{s}_{then}(y) \wedge y \notin \overline{s}_{else}) \vee (y \notin \overline{s}_{then} \wedge v = \overline{s}_{else}(y))\} \\ \overline{x}' = \overline{x}[y \mapsto v]\ \forall y \in \overline{x} : v\ = \overline{x}_{then}(y) = \overline{x}_{else}(y) \end{array}}{\begin{array}{c} G \vdash \langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{\texttt{void}\}, (\overline{x}_{then}, \overline{s}_{then}), (\overline{x}_{else}, \overline{s}_{else}), (\overline{x}, \overline{s})\rangle \\ \rightarrow_{\text{sc}} \\ G \vdash \langle \texttt{void}, (\overline{x}', \overline{s}_2)\rangle \end{array}}$$

(If-sbool-phi)

Figure 1.10: Sample of server-side semantics for secret conditional commands

in both branches then it is a potential security threat and therefor we make it an execution error and the transition system is "stuck". In the next section we will define a type system, which will prevent well-typed SMCL programs to get "stuck".

## 1.5 Static Semantics

The main goal of the SMCL design is to provide the programmer with a set of well-defined security guarantees. In this Section we present a type system, which provides two guarantees, well-typed programs do *not get "stuck"* and well-typed programs are *hoistable*. The two guarantees will be fundamental parts of showing that SMCL programs are secure in Section 1.6. A *well-typed* program is a program which can be assigned a type by the type system.

Well-typed programs do not get "stuck". This is a standard safety condition [73] which guarantees that well-typed programs do not end in a configuration, which is a not a terminal configuration and no evaluation rule apply. This safety condition should apply to client definitions as well as to server definitions.

Programs must be *hoistable*, which intuitively means that there are no side-effects in the computations defined in branches of secret conditionals. Side-effects in secret conditionals like I/O or assignment to public variables (implicit flow [36]) are a security threat and we forbid it by considering it a programmer error.

We start by defining hoistability and then present a type system which conservatively approximates hoistability and prevents SMCL programs from getting "stuck".

### 1.5.1 Hoistability

An expression is said to be inside a conditional if it syntactically occurs in either branch of the conditional. An expression inside a secret conditional is *hoistable* if it can be moved (hoisted) immediately before the conditional in the control flow graph without changing the result of evaluating the conditional.

A variable is *locally declared* in a conditional if its declaration int $x = \ldots$ is inside the conditional. A variable is called *public* if it is declared to be of a public type, e.g. bool or int.

A secret conditional is said to be hoistable if every syntacically contained I/O operation, function call, communication, and assignment to a non-locally declared variable are hoistable. A program is hoistable if all syntactically contained secret conditionals are hoistable.

A fragment of a program containing a hoistable assignment is shown in Figure 1.11 (left) where the assignment $x = y$ can be hoisted without changing the result of evaluating the fragment. Note that the conditional is hoistable even with the code int $h = y$ because the $h$ is locally declared in the then branch. The program fragment on the right of Figure 1.11 shows an example of a nonhoistable conditional, because $x = z$ only occurs in one of the branches.

```
sbool b = true;
sint s = 0;
int x = 7;
int y = 7;
if (b) {
  x = y;
  int h = y;
  s = s + h;
}
else {
  x = y;
  s = s + 1;
}
```

```
sbool b = true;
sint s = 0;
int x = 7;
int z = 7;
if (b) {
  x = z;
  s = s + x;
}
else {
  s = s + 1;
}
```

Figure 1.11: (Left) A hoistable conditional and (right) a non-hoistable conditional

Hoistability is a non-trivial property and is thus clearly undecidable by Rice's theorem [76]. Hoistability is implied by conventional requirements for noninterference [88, 101, 103]. Instead of fixing a specific decidable requirement, we will allow

the implementation of the SMCL compiler to perform any sound and conservative approximation of this property.

In the rest of this section we will present a conservative approximation to ensure that all SMCL programs are hoistable.

### 1.5.2 A Type System for Hoistability

SMCL is a strongly typed imperative language with a broad range of values as shown in Figure 1.12. The figure shows the *fundamental types* which include integers, booleans, clients, strings, tuples - where $\ell$ denotes the label, arrays, and void.

| | | | |
|---|---|---|---|
| $\tau$ | ::= | bool | booleans |
| | \| | int | integers |
| | \| | client | client identity |
| | \| | string | strings |
| | \| | $[\ \ell_i : (\tau_i, \rho_i)^{i \ldots n}\ ]$ | tuples |
| | \| | $\tau[]$ | arrays |
| | \| | void | void |

Figure 1.12: Different types of values in the language

In SMCL the programmer is required to provide type annotations on declarations of local variables, field variables, and formal parameters. The annotations describe both the fundamental type and the *security level* of values to be contained in the variable, field, or returned by a function. We extend the fundamental types with a notion of security types [78] based on a lattice of security levels as introduced and used by many others before [36–38, 101, 103]. We use a lattice of two security levels, the secret or high-level, $\mathsf{S}$, and the public or low-level, $\mathsf{P}$, where $\mathsf{P} \sqsubseteq \mathsf{S}$. The type system can be generalized to an arbitrary lattice since we don't use the fact that there are only two levels. As an example consider the program in Figure 1.13 where the variable $x$ is declared to be of type sint, which indicates that the values of $x$ are integers and their security level is secret. We denote these values as being secret integers. Similar we define (public) integers of public security level like $y$ as int. Arrays and tuples can only be public in SMCL, this is enforced by only providing public type annotations for those values. For information about the challenges and possible solutions to secret pointers to tuples see the work by Banerjee et al. [9, 10].

```
sint x = 7;
int y = 0;
if (x > 12) {
  x = 1;
}
else {
  y = 0;
}
y = 0;
```

Figure 1.13: Example program

Types at the secret security level are mainly used in server definitions, where they are used to prevent insecure flow of information. In client definitions we only use security types to control the security level of values returned from the server (annotations on function returns and tunnels). The type system is naturally divided into two parts: the client-side type system for client definitions and the server-side type system for server definitions. The client-side type system is mainly standard and we will not go into details with it here. In the following we will only focus on the server-side type system. We refer to [70] for a complete account of the type system.

The server-side type system is designed to enforce the hoistability property and is based on the type system by Volpano and Smith [101] which includes *effects* in the style of Jouvelot and Gifford [53]. In [101] noninterference is established by recording and preventing assignment side-effects at the appropriate places like no assignments to public variables in secret conditionals. We modify this type system to enforce hoistability by tracking any potentially side-effecting operations in SMCL, e.g. assignments and I/O. Our type system is more permissive by allowing assignments to locally defined public variables in secret conditionals. The tracking is done by recording different information for *functions*, *commands*, *expressions*, and *variables*. In particular we will be recording information about assignments and I/O for each command occurring in the program. Informally the tracking is done by observing the "lowest" security level at which an assignment has been done at each command. In this way we can ensure that no assignments to public variables occur in secret conditionals.

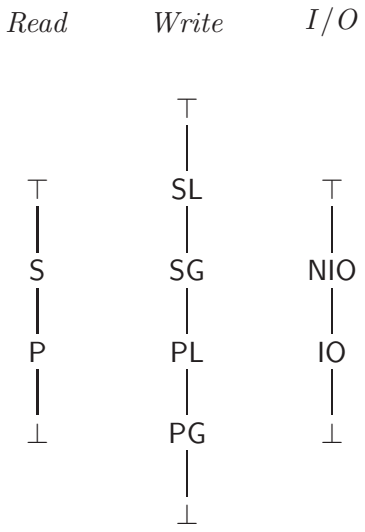



Figure 1.14: Read, write, and I/O Lattices

Formally it gets a bit more involved because we need to record the highest security level at which a variable has been *read*, and second we record the least security level at which a variable has been *written to* and whether the variable is locally declared.

We use the *Read* lattice shown in Figure 1.14 with the two levels S and P to describe the variables read. This is similar to Volpano and Smith [101, 103] and Denning [36] with the addition that $\bot$ means that no variable has been read. We use $\rho$ to denote elements of the *Read* lattice. I/O is captured in a similar fashion to reading of variables with the two level *I/O* lattice of possible I/O IO and definitely no I/O NIO. We use $\iota o$ to denote elements of the *I/O* lattice.

We use the *Write* lattice also shown in Figure 1.14 to decide the security level at which a variable has been written to, and whether the variable is locally declared. The elements of the lattice are: a given variable is secret and locally declared $\mathsf{SL}$, secret and non-locally declared $\mathsf{SG}$, public and locally declared $\mathsf{PL}$, public and non-locally declared $\mathsf{PG}$, and $\top$ means "no variable written", its usefulness will become clear later. There is the partial relation $\mathsf{PG} \sqsubseteq \mathsf{PL} \sqsubseteq \mathsf{SG} \sqsubseteq \mathsf{SL} \sqsubseteq \top$ on the elements. We use $\nu$ to denote elements of the *Write* lattice.

We use the *Read*, *Write*, and *I/O* lattices along with the fundamental types as foundation for the actual types, which are stratified into four kinds:

- Variable types, $(\tau, \rho)$-var
- Expression types, $(\tau, \rho, \nu, \iota o)$-exp
- Command types, $(\nu, \iota o)$-cmd
- Function types, $(\nu, \iota o)$-$(\tau_0, \rho_0)$-fun$((\tau_1, \rho_1), \ldots, (\tau_n, \rho_n))$

The *variable types* $(\tau, \rho)$-var are given to variables. A variable type describes the kind $\tau$ and the security level $\rho$ of the values which may be assigned to the variable. The security levels are elements from the *Read* lattice, which we also call the *Read* types, similar for the other lattices, the *Write* types and the *I/O* types.

*Expression types* $(\tau, \rho, \nu, \iota o)$-exp are the types assigned to expressions. Intuitively an expression has a given type $(\tau, \rho, \nu, \iota o)$-exp if the values computed by the expression have fundamental type $\tau$, there will be a read from variables of at most security level $\rho$, assignments to variables of at least level $\nu$, and whether there is potential for any I/O $\iota o$ during the execution of the expression.

*Command types* are assigned to commands. A command with type $(\nu, \iota o)$-cmd is expected to maintain the invariant that no assignments are made to a variable in the command of type lower than $\nu$ and $\iota o$ describes whether I/O may occur during the execution of the command.

*Function types* $(\nu, \iota o)$-$(\tau_0, \rho_0)$-fun$((\tau_1, \rho_1), \ldots, (\tau_n, \rho_n))$ are assigned to functions. A function has a declared *return type* $(\tau, \rho_0)$ along with a number of *argument types* $(\tau_i, \rho_i)$. Both the return type and the argument types consists of a fundamental type and a read type with similar semantics as variable types. The read type of the return type indicates that a read of a variable of at most security level $\rho_0$ in the computation of any return value. These types are treated in the standard [73] way and are covariant and contravariant with respect to subtyping respectively. Furthermore function types have an additional component which captures if there will not be assignments of type lower than $\nu$ and $\iota o$ describes whether there might be any I/O occurring during the evaluation of the function. Function types are monomorphic and not first-order in the security levels as in [101] which simplifies our treatment of functions.

The partial ordering $\sqsubseteq$ of lattice elements give rise to a straight forward reflexive, transitive, and anti-symmetric subtyping relation $\leq$ when we view the elements of the three lattices as types. An example is given in Figure 1.15 for the *Read* lattice. We use these subtype relations as basis for subtype relations on variable types, expression types, command types, and function types. In Figure 1.16 we define the subtyping relation for each of the types. We show only the full set of rules for variable types since all of the rules except for the base case are similar. Variable types are covariant in the fundamental type and the *Read* type. Expression types are also covariant in the fundamental type, the *Read* type, and the *I/O* type, but contravariant in the *Write* type due to the difference in the requirement; the *Read* types record the highest level read, whereas the *Write* types record the lowest level assigned. Command types are contravariant in the *Write* type and covariant in the *I/O* type. Function types are contravariant in the arguments and *Write* type and covariant in the return type and the *I/O* type.

The least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operator on the lattices give rise to two useful operations on the subtype relation: The least common super type $\veebar$ and the greatest common subtype $\barwedge$. The operations naturally extends from the binary to n-ary case and we define $\veebar_{i=1}^{n}\rho_i$ as $((\rho_1 \veebar \rho_2) \veebar \ldots) \veebar \rho_n)$ and similar for $\barwedge_{i=1}^{n}\rho_i$ and similar for the other types.

$$\frac{\rho \sqsubseteq \rho'}{\rho \leq \rho'}\ (\rho\text{-Base}) \qquad \rho \leq \rho\ (\rho\text{-Reflexivity})$$

$$\frac{\rho \leq \rho' \qquad \rho' \leq \rho''}{\rho \leq \rho''}\ (\rho\text{-Transitive}) \qquad \frac{\rho \leq \rho' \qquad \rho' \leq \rho}{\rho = \rho'}\ (\rho\text{-Anti-symmetric})$$

Figure 1.15: Example subtype relation for elements of the *Read* lattice viewed as types

We now present the three most interesting type rules of the type system. The other type rules are mainly as one would expect, See [70] for a complete treatment. In Figure 1.18 we present the type rule for assignment and in Figure 1.19 we show the two type rules for conditionals. We start by describing the typing context $\Gamma_t = [\Sigma, \eta, \mu, \varpi]$. The typing context is a tuple of the server declaration $\Sigma$, the var-typing $\eta$, the write-typing $\mu$, and the return type of the current function $\varpi$. The var-typing and the write-typing are finite maps from variables to variable-types and write types respectively.
A typing judgment has the form $\Gamma_t \vdash C\ :\ (\nu, \iota o)$-cmd for commands. The judgment means that command $C$ has type $(\nu, \iota o)$-cmd, assuming that $\Gamma_t$ prescribes the server declaration, types for any variable in $C$ and a return type for the current function. Similarly for variable, expression, and function types. The server declaration contains information from which we can easily construct the type of tuples and functions (arguments and return type).

The rule for assignment ensures no explicit flow [36] ($\rho' \leq \rho$ ensures that a secret value cannot be assigned to a public variable) and no assignment of values of incompatible types ($\tau'$ must be a subtype of $\tau$).

We use the write function shown in Figure 1.17 to allow assignments to public variables if they are declared within the same block as the assignment occurs. The function "write" computes the security level and locality of a given variable, read-typing, and program point $p$. We use a variant of a *reaching definitions* analysis [72] to compute the variable declarations that are local at a given point. The set LRD(p) contains the locally declared variables at the program point p. A variable declaration is locally declared with respect to a program point if the declaration is in the same block of the code as the program point. The body of a function, conditional, and while-, and for-loop define blocks of code.

There are two typing rules for conditional commands. The first rule TIf-secret define the type of conditional commands with a secret condition and the second rule TIf-public the type of conditionals with public condition. The rule TIf-public just propagate the side-effects of the condition and the branches, whereas the rule TIf-secret is more interesting. Here we require no I/O NIO in the branches and assignments must be on variables of type at least public and local ($\mathsf{PL} \leq \nu_j\ j \in \{1, 2\}$). This prohibits implicit flow to public variables declared outside the branches, but allows assignment to variables declared inside the branches. The I/O of the condition determines the I/O of the conditional. The variables written to is the greatest common subtype of those in the branches and in the condition ($\barwedge_{i=0}^{2}\nu_i$).

$$\frac{\tau \leq \tau' \qquad \rho \leq \rho'}{(\tau,\rho)\text{-var} \leq (\tau',\rho')\text{-var}} \quad \text{(Var-base)}$$

$$(\tau,\rho)\text{-var} \leq (\tau,\rho)\text{-var} \quad \text{(Var-reflexivity)}$$

$$\frac{(\tau,\rho)\text{-var} \leq (\tau',\rho')\text{-var} \qquad (\tau',\rho')\text{-var} \leq (\tau'',\rho'')\text{-var}}{(\tau,\rho)\text{-var} \leq (\tau'',\rho'')\text{-var}} \quad \text{(Var-transitive)}$$

$$\frac{(\tau,\rho)\text{-var} \leq (\tau',\rho')\text{-var} \qquad (\tau',\rho')\text{-var} \leq (\tau,\rho)\text{-var}}{(\tau,\rho)\text{-var} = (\tau',\rho')\text{-var}} \quad \text{(Var-antisymmetric)}$$

$$\frac{\tau \leq \tau' \qquad \rho \leq \rho' \qquad \nu' \leq \nu \qquad \iota o \leq \iota o'}{(\tau,\rho,\nu,\iota o)\text{-exp} \leq (\tau',\rho',\nu',\iota o')\text{-exp}} \quad \text{(Exp-base)}$$

$$\frac{\nu' \leq \nu \qquad \iota o \leq \iota o'}{(\nu,\iota o)\text{-cmd} \leq (\nu',\iota o')\text{-cmd}} \quad \text{(Com-base)}$$

$$\frac{\begin{array}{c}\rho_i' \leq \rho_i^{\,i\in 1,\ldots,n} \qquad \tau_i' \leq \tau_i^{\,i\in 1,\ldots,n} \\ \nu' \leq \nu \qquad \iota o \leq \iota o' \qquad \rho_0 \leq \rho_0' \qquad \tau_0 \leq \tau_0'\end{array}}{(\nu,\iota o)\text{-}(\tau_0,\rho_0)\text{-fun}((\tau_1,\rho_1),\ldots,(\tau_n,\rho_n)) \leq (\nu',\iota o')\text{-}(\tau_0',\rho_0')\text{-fun}(\rho_1',\ldots,\rho_n')}$$

(Fun-base)

Figure 1.16: Subtype relation on the variable, expression, command, and function types

$$\begin{array}{rcll} \text{write}(x,\,\eta,\,\text{p}) & = & \mathsf{SL} & \text{if } \eta(x) = (\tau,\mathsf{S}) \text{ and } x \in \text{LRD(p)} \\ & = & \mathsf{SG} & \text{if } \eta(x) = (\tau,\mathsf{S}) \text{ and } x \notin \text{LRD(p)} \\ & = & \mathsf{PL} & \text{if } \eta(x) = (\tau,\mathsf{P}) \text{ and } x \in \text{LRD(p)} \\ & = & \mathsf{PG} & \text{if } \eta(x) = (\tau,\mathsf{P}) \text{ and } x \notin \text{LRD(p)} \end{array}$$

Figure 1.17: Write function, returns the locality of $x$

### 1.5.3 Proof of Soundness

In this subsection we prove the soundness of our type system. The proof implies that no terms of the language ever gets "stuck" and hence there is no implicit flow. We use the standard technique of proving a progress theorem and a preservation theorem. To see that the type system ensures hoistability one has to inspect the type system to see that the system propagates information soundly and the only places where hoistability is relevant is in the type rules for assignments and conditionals.

Technically we would need to prove the progress and preservation theorems over the transition system of the SMCL semantics. However, such a proof follows more clearly if we project out the closed terms of the configurations. A closed term $t$ of a configuration is a command or an expressions with no free variables. We will write $t \rightarrow t'$ for the

$$\frac{\Gamma_t \vdash x : (\tau, \rho)\text{-var} \qquad \Gamma_t \vdash e : (\tau', \rho', \nu', \iota o')\text{-exp} \qquad \tau' \leq \tau \qquad \rho' \leq \rho \qquad \nu = \text{write}(x, \eta, x = \text{e})}{\Gamma_t \vdash x = e : (\tau, \rho, \nu \mathbin{\overline{\wedge}} \nu', \iota o')\text{-exp}} \ (\mathsf{TAssign})$$

Figure 1.18: Typing rule for assignment

$$\frac{\Gamma_t \vdash e : (\text{bool}, \mathsf{S}, \nu_0, \iota o)\text{-exp} \qquad \Gamma_t \vdash C_1 : (\nu_1, \mathsf{NIO})\text{-cmd} \qquad \Gamma_t \vdash C_2 : (\nu_2, \mathsf{NIO})\text{-cmd} \qquad \mathsf{PL} \leq \nu_j \ j \in \{1, 2\}}{\Gamma_t \vdash \texttt{if}\ (e)\ \{C_1\}\ \texttt{else}\ \{C_2\} : (\overset{2}{\underset{i=0}{\overline{\wedge}}}\ \nu_i, \iota o)\text{-cmd}} \ (\mathsf{TIf\text{-}secret})$$

$$\frac{\Gamma_t \vdash e : (\text{bool}, \mathsf{P}, \nu_0, \iota o_0)\text{-exp} \qquad \Gamma_t \vdash C_1 : (\nu_1, \iota o_1)\text{-cmd} \qquad \Gamma_t \vdash C_2 : (\nu_2, \iota o_2)\text{-cmd}}{\Gamma_t \vdash \texttt{if}\ (e)\ \{C_1\}\ \texttt{else}\ \{C_2\} : (\overset{2}{\underset{i=0}{\overline{\wedge}}}\ \nu_i, \overset{2}{\underset{i=0}{\underline{\vee}}}\ \iota o_i)\text{-cmd}} \ (\mathsf{TIf\text{-}public})$$

Figure 1.19: Typing rules for conditional commands

closed terms $t$ and $t'$ projected from the configurations $\gamma, \gamma'$ where $\gamma \to \gamma'$. We use $T$ to range over all four kinds of types.

**Theorem 1.1** (Progress). *$t : T$ for some type $T$ then either $t$ is a value or else there is some $t'$ with $t \to t'$.*

*Proof.* By induction on the derivation of $t : T$, we only show the interesting cases for the conditional command:

[Case TIf-Secret] $t = \texttt{if}\ (t_1)\ \{t_2\}\ \texttt{else}\ \{t_3\}$

$$t_1 : (\text{bool}, \mathsf{S}, \nu_0, \iota o)\text{-exp} \quad t_2 : (\nu_1, \mathsf{NIO})\text{-cmd} \quad t_3 : (\nu_2, \mathsf{NIO})\text{-cmd}$$

$t$ is clearly not a value, so we need to show that there is some $t'$ with $t \to t'$. By the induction hypothesis $t_1$ is either a value or there is a term $t_1'$ such that $t_1 \to t_1'$. If $t_1$ is not a value then rule If-stm applies with $t' = \texttt{if}\ (t_1')\ \{t_2\}\ \texttt{else}\ \{t_3\}$.

If $t_1$ is a value then it is either $\boxed{true}$ or $\boxed{false}$. In either case exactly one of the rules If-SBool-* applies because TIf-Secret ensures that $\nu_j \geq \mathsf{PL}$ for both branches. The result is the evaluation of both $t_2$ and $t_3$.

[Case TIf-Public] $t = \texttt{if}\ (t_1)\ \{t_2\}\ \texttt{else}\ \{t_3\}$

$$t_1 : (\text{bool}, \mathsf{P}, \nu_0, \iota o)\text{-exp} \quad t_2 : (\nu_1, \mathsf{IO}_1)\text{-cmd} \quad t_3 : (\nu_2, \mathsf{IO}_2)\text{-cmd}$$

$t$ is clearly not a value, so we need to show that there is some $t'$ with $t \to t'$. By the induction hypothesis $t_1$ is either a value or there is a term $t_1'$ such that $t_1 \to t_1'$. If $t_1$ is not a value then rule If-stm applies with $t' = \texttt{if}\ (t_1')\ \{t_2\}\ \texttt{else}\ \{t_3\}$.

If $t_1$ is a value then it is either `true` or `false`. In either case the usual rule for conditionals applies evaluating either $t_2$ or $t_3$.

□

**Theorem 1.2** (Preservation). *If* $\Gamma_t \vdash t : T$ *and* $t \to t'$*, then* $\Gamma_t \vdash t' : T$*.*

*Proof.* By induction on the derivation of $t : (\nu, \iota o)$-cmd, we only show the interesting cases for conditional commands:

[Case TIf-Secret] $t = \texttt{if}\ (t_1)\ \{t_2\}\ \texttt{else}\ \{t_3\}$

$$t_1 : (\text{bool}, \mathsf{S}, \nu_0, \iota o)\text{-exp} \quad t_2 : (\nu_1, \mathsf{NIO})\text{-cmd} \quad t_3 : (\nu_2, \mathsf{NIO})\text{-cmd}$$

From the evaluation rules for secret conditionals we see that $t'$ can be one of the following terms (there are four more possibilities but for brevity we only show the cases we also mentioned in Section 1.4):

[Case If-stm] $t' = \texttt{if}\ (t_1')\ \{t_2\}\ \texttt{else}\ \{t_3\}$, where by induction we know that $t_1 \to t_1'$ and $t_1' : (\text{bool}, \mathsf{S}, \nu_0, \iota o)$-exp. Thus by TIf-Secret $t' : T$

[Case If-sbool-then-final] $t_1 = \boxed{v}$ and $t' = \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{t_3\}$, where by induction we know that $t_2 \to \texttt{void}$ and $\texttt{void} : T$. Thus by TIf-Secret $t' : T$

[Case If-sbool-else] $t_1 = \boxed{v}$, $t_2 = \texttt{void}$, and $t' = \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{t_3\}$, where by induction we know that $t_3 \to t_3'$ and $t_3' : T$. Thus by TIf-Secret $t' : T$

[Case If-sbool-phi] $t_1 = \boxed{v}$, $t_2 = t_3 = \texttt{void}$, and $t' = \texttt{void}$. By TIf-Secret $\nu_j \geq \mathsf{PL}$ and because there is no I/O in the evaluation of rule If-sbool-phi and the assignments respect the assignments made in each branch then $t' : T$

[Case TIf-Public] $t = \texttt{if}\ (t_1)\ \{t_2\}\ \texttt{else}\ \{t_3\}$

$$t_1 : (\text{bool}, \mathsf{P}, \nu_0, \iota o)\text{-exp} \quad t_2 : (\nu_1, \iota o_1)\text{-cmd} \quad t_3 : (\nu_2, \iota o_2)\text{-cmd}$$

From the evaluation rules for secret conditionals we see that $t'$ can be one of the following terms (here we also leave out the two cases we did not cover in Section 1.4):

[Case If-stm] $t' = \texttt{if}\ (t_1')\ \{t_2\}\ \texttt{else}\ \{t_3\}$, where by induction we know that $t_1 \to t_1'$ and $t_1' : (\text{bool}, \mathsf{S}, \nu_0, \iota o)$-exp. Thus by TIf-Secret $t' : T$

□

### 1.5.4 Termination

SMCL programs may be nonterminating due to infinite `while` loops or infinite recursions. Non of these can of course depend on secret values, because that would leak information about the secret values. The type system above presents `while` loops which depends on secret values. Recursion on secret values would need to involve a secret conditional to test if the recursion should halt or continue. Since we always execute both branches, the execution would never halt, and we consider it a programming error to have nonterminating code in a branch of a secret condition. We require all SMCL programs to be *branch terminating*:

**Definition 1.1** (Branch terminating). *An SMCL program is branch terminating if both branches of every secret conditional terminates.*

We enforce branch termination by a static analysis that conservatively checks the branches for termination. There are a number of termination analysis [60] which can be adopted to SMCL, however we chosen to use a simple syntactic criteria in the present implementation of the SMCL compiler for simplicity. We have yet to write examples where a more advanced analysis is needed. The criteria prevents `return` commands, I/O, `while` loops, and function calls from occurring in branches of secret conditionals.

## 1.6 Security Guaranteed

In this Section we will define what it means for an SMCL program to be secure, and prove that all well-typed SMCL programs are secure.

Security in the world of SMC is the protection of information from being revealed explicit or implicitly to any unintended entity. An adversary is an entity (e.g. person or organization) who wants to learn the information. Security is achieved by distributing parts of the information among the server parties, and the information is considered secure if no more than a predefined threshold of the server parties are corrupted by an adversary. If the adversary corrupts more than the threshold of server parties then all security guarantees are off, because the adversary may reconstruct the secret. This is called *threshold* security.

Information is not only distributed among the server parties but is also the input to computations which may be observed by an adversary. The adversary may be able to deduce information about the inputs and the intermediate results by observing the side-effects of the computation, if the side-effects depends on the secret information. The side-effects may be *physical* side-effects like time, electric power consumption, electromagnetic radiation, noninterference, or interaction with cache, harddrive, or other physical objects. Information may also be leaked through logical or *semantic* side-effects like deducing the value of a variable from some other but related values which has all ready been made public, e.g. say $x$ is containing a secret value, we make $x\%10$ and $x/10$ public then it is easy for an adversary to compute $x$.

Security in SMCL must ultimately be based on threshold security and prevent physical and semantic side-effects. Threshold security is achieved by using the cryptographic runtime, which must provide us with the following security guarantees:

1. Operations on secret values are free of physical side-effects
2. Operations compute the correct result
3. Operations cannot be compromised unless at least a threshold of the server parties are corrupted

We will prevent physical side-effects by using a carefully crafted semantics and type system. If a computation on secret values has any observable side-effects which depends on its arguments then it is a potential security threat, and history has shown us that someone will devise an attack which exploits this vulnerability. The semantic security is more difficult to ensure, and we only present an aid for the SMCL programmer to avoid semantics leaks in Subsection 1.6.5.

In the rest of this section we will formally state and prove that any computation expressed as an SMCL program is free of physical side-effects.

### 1.6.1 A Model of Security for SMCL

In this section we develop a model of what it will mean for SMCL programs to be free of physical side-effects, and in the next subsection we will prove that it is the case.

First we define the capabilities of the adversary $\mathcal{A}$. His computational power is limited to probabilistic polynomial time computations, which means that he cannot break usual cryptographic algorithms, e.g. he cannot factor a prime number of reasonable size in reasonable time. Besides the computational capabilities the adversary may also make any measurement at each step of the execution, and record the information, and make future measurements based on this. Furthermore the adversary can corrupt any subset (up to a threshold) of clients ($\chi \subseteq \{\kappa_1, \ldots, \kappa_n\}$) and server parties ($\Sigma \subseteq \{\sigma_1, \ldots, \sigma_m\}$). By corrupting a party the adversary takes control of the machines executing the party. The adversary may also control when and if messages among

parties are delivered. But may not control the content of messages from non-corrupted parties. The adversary learns the secret values of the client parties he corrupts, and if he takes control of more than the threshold of server parties then he learns all the secret values - including those submitted by client parties he has not corrupted.

The adversary may be considered in a *static* case or an *adaptive* case. In the static case the adversary decides which set of parties to corrupt before the program execution begin. In the adaptive case, the adversary corrupts parties at will throughout the computation. Adversaries also come in two other kinds of flavors *malicious* (corrupted parties follow arbitrary instruction set by the adversary) and *semi-honest* (parties follows the protocol). We consider an adaptive, semi-honest adversary because this is what the underlying cryptographic runtime supports.

The adversary may make a measurement at each step of the evaluation of an SMCL program to observe the value of one or more physical properties $\phi$. A measurement $m \in \mathcal{M}$ determines the value of a finite set of physical properties $(\phi_1, \ldots, \phi_n)$ up to the precision of the used measuring tool.

**Definition 1.2** *A measurement on a transition is a triple* $(\gamma_\alpha, m, \gamma'_\alpha) \in \Gamma_\alpha \times \mathcal{M} \times \Gamma_\alpha$ *for states* $\gamma_\alpha \rightarrow \gamma'_\alpha \in \Gamma_\alpha$ *and* $m \in \mathcal{M}$*. We write the measurements performed when going from state* $\gamma_\alpha$ *to state* $\gamma'_\alpha$ *as* $\gamma_\alpha \xrightarrow{m}_\alpha \gamma'_\alpha$*.*

To capture our intuitive notion of when measurements should be equivalent we introduce the concept *indistinguishability*. Two measurements $m_1, m_2$ are indistinguishable (written $m_1 \approx m_1$) if the distributions of the measurements are the same. This formulation of physical observations is similar to the one found in [65].

There are three basic assumptions we make about indistinguishability of measurements. The three assumptions are listed in Definition 1.3. The *first* is a property of the representation of secret values. The information obtainable through physical measurements on a configuration (written $\langle t, \overset{m_1}{(\overline{x}, \overline{s})}\rangle$) is indistinguishable from the same measurements made on another configuration $\langle t, \overset{m_2}{(\overline{x}, \hat{\overline{s}})}\rangle$, if the two configurations only differ on secret state and the adversary does not corrupt more than the threshold of server parties. If this was not the case then the storage of secret values would leak information to the adversary, and computation on such values would be of small value.

The *second* assumption informally states that if we do the same thing we will observe the same distribution. If a step of evaluation is made from a configuration $\langle t, (\overline{x}, \overline{s})\rangle$ to a state with unchanged secret state $\langle t', (\overline{x}', \overline{s})\rangle$ then it will never be the case that a different distribution on the measurements is observed if we make the exact same step at any point of the future. This captures the reproducibility of physical results. If the state is a step in the evaluation which determines the value of some physical property by rolling a dice each time it is executed, the property would definitely be of no value to adversary or anybody else.

The *third* assumption informally states that if a transition does not affect the term or state then the measurements on two transitions which only differ on secret state are indistinguishable.

**Definition 1.3** $m_1 \approx m_2$ *if either of the three cases hold:*

1. $$\langle t, \overset{m_1}{(\overline{x}, \overline{s})}\rangle \quad \textit{and} \quad \langle t, \overset{m_2}{(\overline{x}, \overline{s}')}\rangle$$

2. $$\langle t, (\overline{x}, \overline{s})\rangle \xrightarrow{m_1}_\alpha \langle t', (\overline{x}', \overline{s})\rangle$$
$$\langle t, (\overline{x}, \overline{s})\rangle \xrightarrow{m_2}_\alpha \langle t', (\overline{x}', \overline{s})\rangle$$

*3.*

$$\begin{array}{lll} \langle t, (\overline{x}, \overline{s})\rangle & \xrightarrow{m_1}_{\alpha} & \langle t, (\overline{x}, \overline{s})\rangle \\ \left\langle t, (\overline{x}, \hat{\overline{s}})\right\rangle & \xrightarrow{m_2}_{\alpha} & \left\langle t, (\overline{x}, \hat{\overline{s}})\right\rangle \end{array}$$

Any implementation of SMCL must have the property that changing the program counter is indistinguishable. Here we are not referring to which configuration the program counter should be changed to, but the act of changing it. This is a reasonable assumption because changing the configuration is not dependent on the state in any way.

**Definition 1.4** *Changing the program counter is indistinguishable. If two transitions:*

$$\begin{array}{lll} \langle t, (\overline{x}, \overline{s})\rangle & \xrightarrow{m_1}_{\alpha} & \langle t', (\overline{x}', \overline{s}')\rangle \\ \left\langle t, (\overline{x}, \hat{\overline{s}})\right\rangle & \xrightarrow{m_2}_{\alpha} & \left\langle t', (\overline{x}', \hat{\overline{s}}')\right\rangle \end{array}$$

*are indistinguishable then changing the instruction pointer from* $\langle t, (\overline{x}, \overline{s})\rangle$ *to* $\langle t', (\overline{x}', \overline{s}')\rangle$ *or from* $\left\langle t, (\overline{x}, \hat{\overline{s}})\right\rangle$ *to* $\left\langle t', (\overline{x}', \hat{\overline{s}}')\right\rangle$ *respectively is indistinguishable.*

A measurement $m_i = (\phi_0, \ldots, \phi_n)$ may be separated into two disjoint measurements $m_1' = (\phi_0, \ldots, \phi_j), m_1'' = (\phi_{j+1}, \ldots, \phi_n)$ which we write $m_i = m_i' m_i''$. Physical measurements $m_1 = m_1' m_1''$ and $m_2 = m_2' m_2''$ are indistinguishable if they are component-wise indistinguishable:

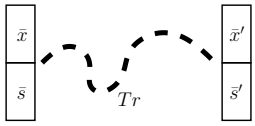
**Definition 1.5** $m_1' m_1'' = m_1 \approx m_2 = m_2' m_2''$ *iff.* $m_1' \approx m_2'$ *and* $m_1'' \approx m_2''$.

The evaluation of an SMCL program under observation by a given adversary leaves a trace of physical properties measured by the adversary.

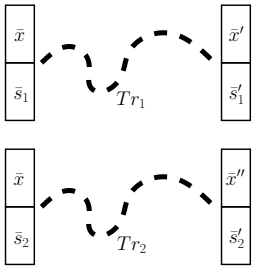
**Definition 1.6** *We define the* physical trace $\mathrm{Tr}(\gamma_\alpha)$ *of a state* $\gamma_\alpha \in \Gamma_\alpha$ *as a finite or infinite sequence of measurements on transitions:* $\mathrm{Tr}(\gamma_\alpha) = (t_1, t_2, \ldots)$ *where* $t_i = \gamma_\alpha^i \xrightarrow{m_i}_{\alpha} \gamma_\alpha^{i+1}$.

**Definition 1.7** *We define the physical trace for the execution of an SMCL program as the physical trace of the start configuration* $\gamma_{start}$ *of the program:* $\mathrm{Tr}(\gamma_{start})$.

We show a physical trace Tr from a state with public state $\overline{x}$ and secret state $\overline{s}$ to one with public state $\overline{x}'$ and secret state $\overline{s}'$ as the illustration:




An SMCL program is intuitively secure if whatever measurements an adversary makes during the execution of a program does not depend on the secret state. Formally if an SMCL program $P$ is executed on public state $\overline{x}$, secret state $\overline{s}_1$, and an adversary makes a sequence of measurements $(m_1, \ldots)$, then he should observe values indistinguishable from the values he would have observed if he had executed the program with the same public state $\overline{x}$, different secret state $\overline{s}_2$, and made the same sequence of measurements. This property can be visualized by the following illustration:




Where the traces are equal $\mathrm{Tr}_1 = \mathrm{Tr}_2$. We call this property the *identity property*. The identity property states that computation does not reveal information about secret state. Formally we start by defining what it means for two configurations to be equivalent:

**Definition 1.8** (Equivalence of configurations). *$\gamma_\alpha \sim \delta_\alpha$ if $\gamma_\alpha = G \vdash \langle t, (\overline{x}, \overline{s})\rangle$, $\delta_\alpha = G' \vdash \langle t, (\overline{x}, \overline{s}')\rangle$, and*

1. *$\gamma_\alpha \xrightarrow{m_1}_\alpha \gamma'_\alpha \Rightarrow \exists\ \delta'_\alpha : \delta_\alpha \xrightarrow{m_2}_\alpha \delta'_\alpha$ and $\gamma'_\alpha \sim \delta'_\alpha, m_1 \approx m_2$*
2. *if $\gamma_\alpha \in T_\alpha$ or $\delta_\alpha \in T_\alpha$ then $\gamma_\alpha = \delta_\alpha$.*

*for some pair of secret states $\overline{s}, \overline{s}'$ and closed term $t$.*

Two states are equivalent if they consists of some global store, the same closed term $t$ and a server state with equal public state but potentially different secret state. The adversary should not be able to tell the difference between the evaluation of the two transitions, so we require that each possible transition from one configuration should be matched by exactly one step from the other configuration, effectively defining a strong bisimilation.

The equivalence of physical traces is defined in terms of equivalence of their start configurations, two traces are equal if their start configurations are equivalent:

**Definition 1.9** (Equivalence of physical traces). *$\mathrm{Tr}(\gamma_\alpha) = \mathrm{Tr}(\gamma'_\alpha)$ if and only if $\gamma_\alpha \sim \gamma'_\alpha$.*

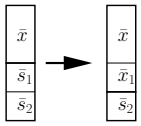
**Definition 1.10** (The Identity Property). *An SMCL program $P$ has the identity property if the trace of the program evaluated on secret state $\overline{s}$ and public state $\overline{x}$ is equal to the trace of the same program evaluated on some (other) secret state $\overline{s}'$ and the same public state $\overline{x}$:*

$$\mathrm{Tr}((\overline{x}, \overline{s})) = \mathrm{Tr}((\overline{x}, \overline{s}'))$$

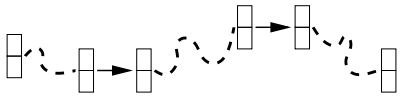
*Note that the secret state may change during the computation, e.g. as a result of client inputs, which is not a problem because the identity property ensures that the trace does not change*

The identity property only holds in general on traces without *open* operations, because an *open* operation explicitly influences the public state which may alter the control flow.

A transition where secret state $\overline{s}_1$ is made public using the *open* operation to become public state $\overline{x}_1$ can be illustrated by:



A complete computation that occasionally makes use of the *open* operation for downgrading can then be described by an alternating sequence of physical traces and these transitions:



The adversary may of course also make measurements during the evaluation of the *open* operation which is modeled in the same way as other transitions.

*Trace security* can now be defined similarly to [63, 64] in terms of *trace sequences*:

**Definition 1.11** (Trace sequence). *A* trace sequence *is a possible infinite sequence of maximal physical traces without open operations* $(\mathrm{Tr}_1((\overline{x}_1, \overline{s}_1)), \ldots)$*, if* $\gamma_\alpha \xrightarrow{m}_\alpha \gamma'_\alpha$ *is an evaluation of an open operation then* $\gamma_\alpha$ *is the last configuration in* $\mathrm{Tr}_i((\overline{x}_i, \overline{s}_i))$*, and* $\gamma'_\alpha$ *is the first configuration in* $\mathrm{Tr}_{i+1}((\overline{x}_{i+1}, \overline{s}_{i+1}))$ *for all* $\mathrm{Tr}_{i+1}((\overline{x}_{i+1}, \overline{s}_{i+1})) \in (\mathrm{Tr}_1((\overline{x}_1, \overline{s}_1)), \ldots)$.

*Trace security* is defined by requiring the identity property to hold for each trace of all the trace sequences of an SMCL program's start configuration and the *open* operation must not reveal any physical observable property which depends on its arguments except for the result.

**Definition 1.12** (Trace security). *An SMCL program P is said to be trace-secure if for all trace sequences* $(\mathrm{Tr}_1, \ldots)$ *initiating from the start configuration of P the following holds:* $\mathrm{Tr}_i((\overline{x}_i, \overline{s}_i)) = \mathrm{Tr}_i((\overline{x}_i, \overline{s'}_i)) \ \forall \ \mathrm{Tr}_i((\overline{x}_i, \overline{s}_i)) \in (\mathrm{Tr}_1((\overline{x}_1, \overline{s}_1)), \ldots)$ *for all sets of secret states* $\overline{s}, \overline{s'}$.

The definition of trace security does not mention the requirements on the *open* operation. This is intentional because by assumption on the runtime the *open* operation has the required properties.

### 1.6.2 Every SMCL Program is Trace-secure

We now proceed by showing that every SMCL program is trace-secure. The security is founded on a solid combination of a type system and well-designed semantics which in combination ensures the necessary properties to prove the main result of trace-security.

The first result we will need to establish is the determinism of the server-side transition systems which will simplify the following proofs a great deal. The Server-Client transition systems is not deterministic by construction, however we can restrict our selves to only consider the server-side part of the SMCL traces.

**Lemma 1.1** *The server-side transition systems are deterministic.*

*Proof.* By inspection of the semantics we see that for each configuration there is at most one rule which applies and that each rule results in exactly one configuration. □

We continue by proving a lemma which informally says that server-side expressions are secure. By secure we mean that the secret state does not influence the public state and physical measurements are indistinguishable for states which only differ on secret state.

**Lemma 1.2** *For all well-typed and non-open configurations in the server-side expression transition system:* $\gamma_{\mathrm{se}}, \gamma'_{\mathrm{se}}, \delta_{\mathrm{se}}$ *where* $\gamma_{\mathrm{se}} \xrightarrow{m_1}_{\mathrm{se}} \gamma'_{\mathrm{se}}$ *and* $\gamma_{\mathrm{se}} \sim \delta_{\mathrm{se}}$ *there exists* $\delta'_{\mathrm{se}}, m_2$ *such that* $\delta_{\mathrm{se}} \xrightarrow{m_2}_{\mathrm{se}} \delta'_{\mathrm{se}}$, $\gamma'_{\mathrm{se}} \sim \delta'_{\mathrm{se}}$, *and* $m_1 \approx m_2$.

*Which can be illustrated by the following diagram:*

$$\begin{array}{ccc} \gamma_{\text{se}} & \sim & \delta_{\text{se}} \\ \downarrow m_1 & & \downarrow m_2 \\ \gamma'_{\text{se}} & \sim & \delta'_{\text{se}} \end{array}$$

*Proof.* Let $\gamma_{\text{se}}, \gamma'_{\text{se}}, \delta_{\text{se}}$ be non-*open* configurations in well-typed in the server-side expression transition system. And let physical measurement $m_1$ be given such that $\gamma_{\text{se}} \xrightarrow{m_1}_{\text{se}} \gamma'_{\text{se}}$ and $\gamma_{\text{se}} \sim \delta_{\text{se}}$. We now prove by induction in the derivation of an expression $e$ that there exists a $\delta'_{\text{se}}, m_2$ such that $\delta_{\text{se}} \xrightarrow{m_2}_{\text{se}} \delta'_{\text{se}}$, $\gamma'_{\text{se}} \sim \delta'_{\text{se}}$, and $m_1 \approx m_2$.

We split the proof into cases for each rule involving server-side expressions, which is possible because the transition system is deterministic by Lemma 1.1. For brevity we only show three representative cases Op-left, Op-sexp, and Assign-exp

[Case Op-left] The structure of the rule is:

$$\frac{G \vdash \langle e_1, \sigma.S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle e'_1, \sigma.S'\rangle}{G \vdash \langle e_1 \ op \ e_2, \sigma.S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle e'_1 \ op \ e_2, \sigma.S'\rangle} \quad \text{(Op-left)}$$

We can identify the following configurations:

$$\begin{array}{rcl} \gamma_{\text{se}} & = & G \vdash \langle e_1 \ op \ e_2, (\overline{x}, \overline{s})\rangle \\ \gamma'_{\text{se}} & = & G \vdash \langle e'_1 \ op \ e_2, (\overline{x}', \overline{s}')\rangle \\ \delta_{\text{se}} & = & G' \vdash \left\langle e_1 \ op \ e_2, (\overline{x}, \hat{\overline{s}})\right\rangle \end{array}$$

For there to be a transition from $\delta_{\text{se}} \xrightarrow{m_2}_{\text{se}} \delta'_{\text{se}}$ then $\delta'_{\text{se}}$ must have the form:

$$\delta'_{\text{se}} = G \vdash \langle e'_1 \ op \ e_2, (\overline{x}'', \overline{s}'')\rangle$$

for public and secret state $\overline{x}'', \overline{s}''$, because of rule Op-left.

By the inductive hypothesis we get that $G \vdash \langle e_1, (\overline{x}, \overline{s})\rangle \xrightarrow{m'_1}_{\text{se}} G \vdash \langle e'_1, (\overline{x}', \overline{s}')\rangle$ implies the existence of a configuration $G \vdash \left\langle e'_1, (\overline{x}', \hat{\overline{s}}')\right\rangle$ with the relationship $G \vdash \left\langle e_1, (\overline{x}, \hat{\overline{s}})\right\rangle \xrightarrow{m'_2}_{\text{se}} G \vdash \left\langle e'_1, (\overline{x}', \hat{\overline{s}}')\right\rangle$ which together with rule Op-left gives that $\overline{x}'' = \overline{x}'$. Furthermore $m'_1 m''_1 = m_1 \approx m_2 = m'_2 m''_2$ due to Definition 1.5 and because we only increment the program counter (captured by $m''_i$) in rule Op-left which is independent of the state.

[Case Assign-exp] The structure of the rule is:

$$\frac{(\overline{x}', \overline{s}') = (\overline{x}, \overline{s})[y \mapsto w]}{G \vdash \langle y = w, (\overline{x}, \overline{s})\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle w, (\overline{x}', \overline{s}')\rangle} \quad \text{(Assign-exp)}$$

We can identify the following configurations:

$$\begin{array}{rcl} \gamma_{\text{se}} & = & G \vdash \langle y = w, (\overline{x}, \overline{s})\rangle \\ \gamma'_{\text{se}} & = & G \vdash \langle w, (\overline{x}', \overline{s}')\rangle \\ \delta_{\text{se}} & = & G' \vdash \left\langle y = w, (\overline{x}, \hat{\overline{s}})\right\rangle \end{array}$$

For there to be a transition from $\delta_{\text{se}} \xrightarrow{m_2}_{\text{se}} \delta'_{\text{se}}$ then $\delta'_{\text{se}}$ must have the form:

$$G \vdash \langle w, (\overline{x}'', \overline{s}'')\rangle$$

for public and secret state $\overline{x}'', \overline{s}''$, because of rule Assign-exp.

We do a case analysis on the value $w$. Either $w$ is a public value $v$ or it is a secret value $\boxed{v}$.

If $w = v$ then the type system guarantees that the variable $y$ will be in the public part of the state, and so only the public state $\overline{x}$ will be modified. Thus $\overline{x}' = \overline{x}[y \mapsto v]$ which applies equally to $\langle e, (\overline{x}, \overline{s})\rangle$ and $\langle e, (\overline{x}, \hat{\overline{s}})\rangle$ thus $\overline{x}'' = \overline{x}'$. The secret state does not change in this case and we get $m_1 \approx m_2$ by assumption (2) of Definition 1.3.

If $w = \boxed{v}$ then the type system guarantees that the variable $y$ will be in the secret part of the state, and so only the secret state $\overline{s}$ will be modified. Thus $\overline{x}'' = \overline{x}' = \overline{x}$. $m_1 \approx m_2$ because variable assignment is independent of the secret value by assumption of the runtime.

[Case Op-sexp] The structure of the rule is:

$$\frac{op(\boxed{v_1}, \boxed{v_2}) = \boxed{v_3}}{G \vdash \langle \boxed{v_1}\ op\ \boxed{v_2}, \sigma.S\rangle \ \rightarrow_{\text{se}}\ G \vdash \langle \boxed{v_3}, \sigma.S\rangle} \quad \text{(Op-sexp)}$$

We can identify the following configurations:

$$\begin{aligned} \gamma_{\text{se}} &= G \vdash \langle \boxed{v_1}\ op\ \boxed{v_2}, (\overline{x}, \overline{s})\rangle \\ \gamma'_{\text{se}} &= G \vdash \langle \boxed{v_3}, (\overline{x}', \overline{s}')\rangle \\ \delta_{\text{se}} &= G' \vdash \langle \boxed{v_1}\ op\ \boxed{v_2}, (\overline{x}, \hat{\overline{s}})\rangle \end{aligned}$$

For there to be a transition from $\delta_{\text{se}} \xrightarrow{m_2}_{\text{se}} \delta'_{\text{se}}$ then $\delta'_{\text{se}}$ must have the form:

$$\delta'_{\text{se}} = G \vdash \langle \boxed{v_3}, (\overline{x}, \hat{\overline{s}})\rangle$$

because of rule Op-sexp and the rule preserves state. $m_1 \approx m_2$ because operations are by assumption of the runtime independent of secret value arguments.

□

Lemma 1.2 will be an important stepping stone in the main theorem, Theorem 1.3 which informally says that all steps in the evaluation of well-typed and branch terminating SMCL programs are independent of secret values. This is not true if the step is the evaluation of an *open* operation, so we require the configurations of the steps to be a non-*open* configuration which is a configuration without *open* operations.

**Theorem 1.3** *For all non-open configurations in well-typed and branch terminating SMCL programs:* $\gamma_{\text{sc}}, \gamma'_{\text{sc}}, \delta_{\text{sc}}$ *where* $\gamma_{\text{sc}} \xrightarrow{m_1}_{\text{sc}} \gamma'_{\text{sc}}$ *and* $\gamma_{\text{sc}} \sim \delta_{\text{sc}}$ *there exists* $\delta'_{\text{sc}}, m_2$ *such that* $\delta_{\text{sc}} \xrightarrow{m_2}_{\text{sc}} \delta'_{\text{sc}}$, $\gamma'_{\text{sc}} \sim \delta'_{\text{sc}}$, *and* $m_1 \approx m_2$.

*Which can be illustrated by the following diagram:*

$$\begin{array}{ccc} \gamma_{\text{sc}} & \sim & \delta_{\text{sc}} \\ \downarrow m_1 & & \downarrow m_2 \\ \gamma'_{\text{sc}} & \sim & \delta'_{\text{sc}} \end{array}$$

*Proof.* Let $\gamma_{\text{sc}}, \gamma'_{\text{sc}}, \delta_{\text{sc}}$ be non-*open* configurations in well-typed and branch terminating SMCL programs and physical measurement $m_1$ be given such that $\gamma_{\text{sc}} \xrightarrow{m_1}_{\text{sc}} \gamma'_{\text{sc}}$ and $\gamma_{\text{sc}} \sim \delta_{\text{sc}}$. We now show by coinduction that there exists a $\delta'_{\text{sc}}, m_2$ s.t. $\delta_{\text{sc}} \xrightarrow{m_2}_{\text{sc}} \delta'_{\text{sc}}$, $\gamma'_{\text{sc}} \sim \delta'_{\text{sc}}$, and $m_1 \approx m_2$.

The proof will be a case analysis on the rules of the various transition systems. Most of the rules we will show have the general structure:

$$\frac{G \vdash \langle C_2, S\rangle \ \xrightarrow{m''_i}_{\text{se}}\ G \vdash \langle C'_2, S'\rangle}{G \vdash \langle C_1, S\rangle \ \xrightarrow{m'_i}_{\text{sc}}\ G \vdash \langle C'_1, S'\rangle}$$

Here we split the result of the physical observation $m_i$ into two components $m_i''$ and $m_i'$, where $m_i''$ will be the part of the result related to evaluating the premiss of the rule and $m_i'$ will be the result related to evaluating the conclusion of the rule.

For brevity we only show the proofs for the most interesting cases: If-stm, If-sbool-then-final, If-sbool-else, and If-sbool-phi:

[Case If-stm] The structure of the rule is:

$$\frac{G \vdash \langle e, S\rangle \ \rightarrow_{\text{se}} \ G \vdash \langle e', S'\rangle}{G \vdash \langle \texttt{if}(e)\ \{C_1\}\ \texttt{else}\ \{C_2\}, S\rangle \ \rightarrow_{\text{sc}} \ G \vdash \langle \texttt{if}\ (e')\ \{C_1\}\ \texttt{else}\ \{C_2\}, S'\rangle}$$

(If-stm)

We can identify the following configurations:

$$\begin{array}{rcl} \gamma_{\text{sc}} & = & G \vdash \langle \texttt{if}(e)\ \{C_1\}\ \texttt{else}\ \{C_2\}, (\overline{x}, \overline{s})\rangle \\ \gamma'_{\text{sc}} & = & G \vdash \langle \texttt{if}(e')\ \{C_1\}\ \texttt{else}\ \{C_2\}, (\overline{x}', \overline{s}')\rangle \\ \delta_{\text{sc}} & = & G' \vdash \left\langle \texttt{if}\ (e)\ \{C_1\}\ \texttt{else}\ \{C_2\}, (\overline{x}, \hat{\overline{s}})\right\rangle \end{array}$$

For there to be a transition from $\delta_{\text{sc}} \stackrel{m_2}{\rightarrow}_{\text{sc}} \delta'_{\text{sc}}$ then $\delta'_{\text{sc}}$ must have the form:

$$\delta'_{\text{sc}} = G \vdash \langle \texttt{if}(e')\ \{C_1\}\ \texttt{else}\ \{C_2\}, (\overline{x}'', \overline{s}'')\rangle$$

for public and secret state $\overline{x}'', \overline{s}''$.

Because $\gamma_{\text{sc}} \sim \delta_{\text{sc}}$, $G \vdash \langle e, (\overline{x}, \overline{s})\rangle \ \stackrel{m_1''}{\rightarrow}_{\text{se}} \ G \vdash \langle e', (\overline{x}', \overline{s}')\rangle$, and $G \vdash \langle e, (\overline{x}, \overline{s})\rangle \sim G \vdash \left\langle e, (\overline{x}, \hat{\overline{s}})\right\rangle$ then by Lemma 1.2 we get that $G \vdash \left\langle e, (\overline{x}, \hat{\overline{s}})\right\rangle \ \stackrel{m_2''}{\rightarrow}_{\text{se}} \ G \vdash \left\langle e', (\overline{x}'', \hat{\overline{s}}')\right\rangle$, where $\overline{x}'' = \overline{x}'$ and $m_1'' \approx m_2''$.

Incrementing the program counter is indistinguishable by Definition 1.4 which gives us that $m_1' \approx m_2'$ which combined with Definition 1.5 gives the result $m_1 = m_1' m_1'' \approx m_2' m_2'' = m_2$. So $\delta_{\text{sc}} \sim \delta'_{\text{sc}}$.

[Case If-sbool-then-final] The structure of the rule is:

$$\frac{G \vdash \langle C_1, U_{then}\rangle \ \rightarrow_{\text{sc}} \ G \vdash \langle \texttt{void}, U'_{then}\rangle}{\begin{array}{c} G \vdash \langle \texttt{if}(\boxed{v})\ \{C_1\}\ \texttt{else}\ \{C_2\}, U_{then}, S\rangle \\ \rightarrow_{\text{sc}} \\ G \vdash \langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{C_2\}, U'_{then}, S, S\rangle \end{array}}$$

(If-sbool-then-final)

We can identify the following configurations:

$$\begin{array}{rcl} \gamma_{\text{sc}} & = & G \vdash \langle \texttt{if}(\boxed{v})\ \{C_1\}\ \texttt{else}\ \{C_2\}, (\overline{x}_{then}, \overline{s}_{then}), (\overline{x}, \overline{s})\rangle \\ \gamma'_{\text{sc}} & = & G \vdash \langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{C_2\}, (\overline{x}'_{then}, \overline{s}'_{then}), (\overline{x}, \overline{s}), (\overline{x}, \overline{s})\rangle \\ \delta_{\text{sc}} & = & G \vdash \left\langle \texttt{if}(\boxed{v})\ \{C_1\}\ \texttt{else}\ \{C_2\}, (\overline{x}_{then}, \hat{\overline{s}}_{then}), (\overline{x}, \hat{\overline{s}})\right\rangle \end{array}$$

For there to be a transition from $\delta_{\text{sc}} \stackrel{m_2}{\rightarrow}_{\text{sc}} \delta'_{\text{sc}}$ then $\delta'_{\text{sc}}$ must have the form:

$$\delta'_{\text{sc}} = G \vdash \left\langle \texttt{if}\ (\boxed{v})\ \{\texttt{void}\}\ \texttt{else}\ \{C_2\}, (\overline{x}''_{then}, \overline{s}''_{then}), (\overline{x}, \hat{\overline{s}}), (\overline{x}, \hat{\overline{s}})\right\rangle$$

for public and secret state $\overline{x}''_{then}, \overline{s}''_{then}$.

Because $\gamma_{\text{sc}} \sim \delta_{\text{sc}}$, $G \vdash \langle C_1, (\overline{x}_{then}, \overline{s}_{then})\rangle \ \stackrel{m_1''}{\rightarrow}_{\text{sc}} \ G \vdash \langle \texttt{void}, (\overline{x}'_{then}, \overline{s}'_{then})\rangle$, and $G \vdash \langle C_1, (\overline{x}_{then}, \overline{s}_{then})\rangle \sim G \vdash \left\langle C_1, (\overline{x}_{then}, \hat{\overline{s}}_{then})\right\rangle$ then by the co-inductive

hypothesis we get that: $G \vdash \langle C_1, (\overline{x}_{then}, \hat{\overline{s}}_{then})\rangle \overset{m_2''}{\rightarrow}_{\text{sc}} G \vdash \left\langle \texttt{void}, (\overline{x}''_{then}, \hat{\overline{s}}'_{then})\right\rangle$ where $\overline{x}''_{then} = \overline{x}'_{then}$ and $m_1'' \approx m_2''$

Incrementing the program counter is indistinguishable by Definition 1.4 which gives us that $m_1' \approx m_2'$ which combined with Definition 1.5 gives the result $m_1 = m_1'm_1'' \approx m_2'm_2'' = m_2$. So $\delta_{\text{sc}} \sim \delta'_{\text{sc}}$.

[Case If-sbool-else] The structure of the rule is:

$$\frac{G \vdash \langle C_2, U_{else}\rangle \ \rightarrow_{\text{sc}} \ G \vdash \langle C_2', U'_{else}\rangle}{\begin{array}{c} G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{C_2\}, U_{then}, U_{else}, S\rangle \\ \rightarrow_{\text{sc}} \\ G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{C_2'\}, U_{then}, U'_{else} S\rangle \end{array}}$$

(If-sbool-else)

We can identify the following configurations:

$$\begin{array}{rcl} \gamma_{\text{sc}} & = & G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{C_2\}, U_{then}, (\overline{x}, \overline{s}), S\rangle \\ \gamma'_{\text{sc}} & = & G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{C_2'\}, U_{then}, (\overline{x}', \overline{s}'), S\rangle \\ \delta_{\text{sc}} & = & G \vdash \left\langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{C_2\}, U_{then}, (\overline{x}, \hat{\overline{s}}), S\right\rangle \end{array}$$

For there to be a transition from $\delta_{\text{sc}} \overset{m_2}{\rightarrow}_{\text{sc}} \delta'_{\text{sc}}$ then $\delta'_{\text{sc}}$ must have the form:

$$\delta'_{\text{sc}} = G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{C_2'\}, U_{then}, (\overline{x}'', \overline{s}''), S\rangle$$

for public and secret state $\overline{x}''_{then}, \overline{s}''_{then}$.

Because $\gamma_{\text{sc}} \sim \delta_{\text{sc}}$, $G \vdash \langle C_2, (\overline{x}_{else}, \overline{s}_{else})\rangle \overset{m_1''}{\rightarrow}_{\text{sc}} G \vdash \langle C_2', (\overline{x}'_{else}, \overline{s}'_{else})\rangle$, and $G \vdash \langle C_2, (\overline{x}_{else}, \overline{s}_{else})\rangle \sim G \vdash \left\langle C_2, (\overline{x}_{else}, \hat{\overline{s}}_{else})\right\rangle$ then by the co-inductive hypothesis we get that: $G \vdash \langle C_2, (\overline{x}_{else}, \overline{s}_{else})\rangle \overset{m_2''}{\rightarrow}_{\text{sc}} G \vdash \left\langle C_2, (\overline{x}''_{else}, \hat{\overline{s}}'_{else})\right\rangle$, where $\overline{x}''_{else} = \overline{x}'_{else}$ and $m_1'' \approx m_2''$.

Incrementing the program counter is indistinguishable by Definition 1.4 which gives us that $m_1' \approx m_2'$ which combined with Definition 1.5 gives the result $m_1 = m_1'm_1'' \approx m_2'm_2'' = m_2$. So $\delta_{\text{sc}} \sim \delta'_{\text{sc}}$.

[Case If-sbool-phi] The structure of the rule is:

$$\frac{\begin{array}{c} \overline{s}_1 = \overline{s}\{ \ [y \mapsto \boxed{v} * \overline{s}_{then}(y) + (1 - \boxed{v}) * \overline{s}_{else}(y)] \ \mid \\ \forall y \in \overline{s} : \overline{s}_{then}(y) = \boxed{v'} \ \wedge \ \overline{s}_{else}(y) = \boxed{v''} \ \} \\ \overline{s}_2 = \overline{s}_1\{[y \mapsto \boxed{v}] \mid \forall y : (v = \overline{s}_{then}(y) \wedge y \notin \overline{s}_{else}) \vee (y \notin \overline{s}_{then} \wedge v = \overline{s}_{else}(y))\} \\ \overline{x}' = \overline{x}[y \mapsto v] \ \forall y \in \overline{x} : v \ = \overline{x}_{then}(y) = \overline{x}_{else}(y) \end{array}}{\begin{array}{c} G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{\texttt{void}\}, (\overline{x}_{then}, \overline{s}_{then}), (\overline{x}_{else}, \overline{s}_{else}), (\overline{x}, \overline{s})\rangle \\ \rightarrow_{\text{sc}} \\ G \vdash \langle \texttt{void}, (\overline{x}', \overline{s}_2)\rangle \end{array}}$$

(If-sbool-phi)

We can identify the following configurations:

$$\begin{array}{rcl} \gamma_{\text{sc}} & = & G \vdash \langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{\texttt{void}\}, (\overline{x}_{then}, \overline{s}_{then}), (\overline{x}_{else}, \overline{s}_{else}), (\overline{x}, \overline{s})\rangle \\ \gamma'_{\text{sc}} & = & G \vdash \langle \texttt{void}, (\overline{x}', \overline{s}')\rangle \\ \delta_{\text{sc}} & = & G \vdash \left\langle \texttt{if } (\boxed{v}) \ \{\texttt{void}\} \ \texttt{else } \{\texttt{void}\}, (\overline{x}_{then}, \hat{\overline{s}}_{then}), (\overline{x}_{else}, \hat{\overline{s}}_{else}), (\overline{x}, \hat{\overline{s}})\right\rangle \end{array}$$

For there to be a transition from $\delta_{\text{sc}} \to \delta'_{\text{sc}}$ then $\delta'_{\text{sc}}$ must have the form:

$$\delta'_{\text{sc}} = G \vdash \langle \texttt{void}, (\overline{x}'', \overline{s}'') \rangle$$

for public and secret state $\overline{x}'', \overline{s}''$ because $\gamma_{\text{sc}} \sim \delta_{\text{sc}}$.
Now $\overline{x}' = \overline{x}''$ because:

$$\overline{x}' = \Big(\overline{x}[y \mapsto v] \ \forall y \in \overline{x} : v \ = \overline{x}_{then}(y) = \overline{x}_{else}(y)\Big) = \overline{x}''$$

If $m_1, m_2$ are what can be observed on transitions: $\gamma_{\text{sc}} \xrightarrow{m_1}_{\text{sc}} \gamma'_{\text{sc}}$ and $\delta_{\text{sc}} \xrightarrow{m_2}_{\text{sc}} \delta'_{\text{sc}}$, then $m_1 \approx m_2$ because arithmetic operations on secret values and environment lookup and modification is indistinguishable by assumption on the cryptographic runtime.

□

The main result trace-security now comes as an easy corollary from Theorem 1.3:

**Corollary 1.1** (Trace security). *For all well-typed and branch terminating SMCL programs $P$, and for all trace sequences $(\text{Tr}_1((\overline{x}_1, \overline{s}_1)), \ldots)$ initiating from the start configuration of $P$ the following must hold:* $\text{Tr}_i((\overline{x}_i, \overline{s}_i)) = \text{Tr}_i((\overline{x}_i, \hat{\overline{s}}_i)) \ \forall \ \text{Tr}_i((\overline{x}_i, \overline{s}_i)) \in (\text{Tr}_1((\overline{x}_1, \overline{s}_1)), \ldots)$ *for all sets of secret states $\overline{s}$ and $\hat{\overline{s}}$.*

*Proof.* Follows immediately from Theorem 1.3. □

### 1.6.3 Interpreting the Theorem

Corollary 1.1 implies that doing computations on secret values does not convey any information about the values computed on. This is because the steps of the computation are independent of secret values. The steps referred to are the control flow of the program which does not change based on secret values, but only changes based on public values.

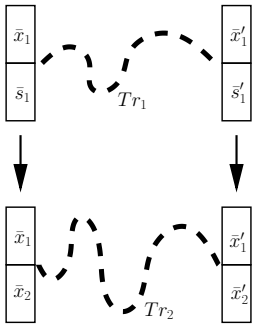
Trace security, i.e. the independence of control flow from secret values makes SMCL programs immune to a wide variety of attacks which exploits the information signaled through variations in traces introduced by secret values. Examples of attacks are (cache) timing attacks, terminations attacks, noninterference attacks, electric power consumption attacks, and electromagnetic radiation attacks.

The variation of traces generally comes from alternatives in control flow introduced by loops, conditions, and recursive functions. SMCL removes the security threats by simply removing the control flow which depends on secret values. SMCL explicitly forbids non-constant loops which depends on secret values, recursive functions cannot have a stop-condition which depends on a secret value because we require that both branches of a secret conditional terminates, and secret conditionals has been made immune to these kinds of attacks by executing both branches. Agat [5] uses a different approach (to avoid timing attacks) by cross-padding the branches with dummy instructions to make sure that their execution time is the same. However this approach is not useful in SMCL because the adversary has access to the instruction pointer of the corrupted parties.

Evaluation of both branches removes timing attacks because the execution time is independent of the condition. Cache-timing attacks are also eliminated because which branch gets executed is independent of the condition, and thus the state of the cache contains no information about the value of the condition.

### 1.6.4 Correctness

As a complement to the identity property we would also like SMCL programs to be *correct*, which is captured by the *commutativity property*. The commutative property states that *open* operations and computations commute:




This property evidently expresses that the secret representation is sound. Note that $\mathrm{Tr}_1$ and $\mathrm{Tr}_2$ will in general clearly be different, but the commutativity property implies that $\mathrm{Tr}_1$ terminates exactly when $\mathrm{Tr}_2$ does.

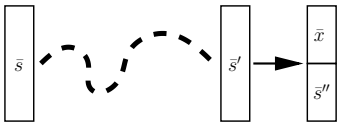
**Definition 1.13** (The Commutativity Property). *An SMCL program $P$ has the commutativity property if the result of the program evaluated on values $\mathcal{E}(P(\overline{x}_1, \overline{s}_1)) = (\overline{x}'_1, \overline{s}_1{}')$ and the same program evaluated on values $\mathcal{E}(P(\overline{x}_1, \overline{x}_2)) = (\overline{x}'_1, \overline{x}'_2)$ where $\overline{x}_2 = open(\overline{s}_1)$ and $\overline{x}'_2 = open(\overline{s}'_1)$. Where $\mathcal{E}$ is the evaluation function defined by the operational semantics in Section 1.4.*

**Theorem 1.4** (The Commutativity Property). *All well-typed SMCL programs have the commutativity property if all the operations provided by the underlying runtime have the commutativity property.*

*Proof.* By coinduction in the semantics of SMCL. □

### 1.6.5 Towards Semantic Security

The security properties provide some basic guarantees about the behavior of SMCL programs. With these guarantees, any computation (without `while` loops) can be made invulnerable to attack by being structured as an *ideal computation*:




Here, all information is kept in secret variables and only at the very end are the outputs $\overline{x}$ made public. However, as shown in Section 1.7, computations on secret values are quite expensive. Thus, a pragmatic computation will keep information in public variables as much as possible without compromising the overall security requirements. The commutativity property ensures that the ideal computation and the pragmatic computation will produce the same output, but the programmer now has the burden of (manually) proving that these two computations will only reveal the same relevant secret information. Since such proofs are difficult to construct, the SMCL compiler provides a simple annotation language to aid the programmer.

The *open* operation should be annotated with secret variables: $open(e|x, y, z)$. The meaning of this annotation is that the programmer recognizes responsibility for compromising the secret values of these variables, and the compiler should check that all compromised variables are mentioned, so the programmer is fully aware of his proof obligations. A program is then only accepted as *well-annotated* if all potential semantic information leaks are explicitly allowed by such annotations. To be conservative, which is a good attitude when security is concerned, any secret variable whose value may have influenced the opened value is viewed as potentially compromised. Thus, for any *open* operation the SMCL compiler computes the set of secret variables that have ever contained a value that may have influenced the value currently being opened. From this set of potentially compromised secret variables we subtract the corresponding sets from all previously executed *open* operations whose values have not since changed. The resulting set of newly compromised secret variables must explicitly be mentioned in the *open* operation. The set of variables which must be mentioned may grow fast thus for ease of annotation we also subtract the variables which may have influence any already mentioned variable. The corresponding analysis is a mixture of a def-use analysis, a liveness analysis, and an available expressions analysis [72]. A simple constant folding analysis also takes care of cases such as multiplying a secret value by the constant zero. This is essentially a bookkeeping procedure where we try to reduce the annotation burden as much as possible. Of course, little is gained if the programmer blindly use these annotations to accept responsibility for the behavior of the pragmatic computation: The idea is that it will be easier to prove equivalence to the ideal computation when the compiler has verified that the program is well-annotated.

## 1.7 Efficiency

Our experiences with SMCR show that Secure Multipart Computations are feasible in practice. However, secret computations are quite expensive as they are based on complex protocols that involve several rounds of communications between the server parties. To illustrate this, we consider a program which computes the sign of a polynomial given coefficients $a$, $b$, and $c$ and a data point $x$. We provide three versions of this program shown in Figure 1.20 and Figure 1.21. To enable proper timings, the client network communications have been replaced with simple assignments. The ideal version keeps everything secret until the output is revealed. The pragmatic version has $x$ as a public value and chooses to allow the value of the polynomial to be public as well as its sign. The public version merely performs an ordinary computation.

In Figure 1.22, we show the timing results from running the compiled versions of these programs on SMCR with 3, 5, and 7 server parties distributed on an equal number of Intel P4 1,8 Ghz with 512 MB of memory (the timings are for one execution of the programs and do not include the time for *preprocessing*, which is a part of the protocols that SMCR uses for multiplications and comparisons). The time needed for preprocessing depends on the number of multiplications done in the computation. The SMCR can be instructed to preprocess a number of multiplications and furthermore use idle time to maintain a pool of preprocessed multiplications. The numbers 1, 2, and 3 denote the threshold that is used in the respective case. Our conclusion is that SMC primitives are expensive but feasible. The slowdown from the public to the pragmatic version is significant but many practical application exists where the slowdown is acceptable. An example is offline auctions where ample time is available for executing the auction. The slowdown from the pragmatic to the ideal version is stunning, but it is to a large extent an unavoidable price for obtaining the full invulnerability of our security properties. In practice, applications will be written in the pragmatic style—making a convincing case for automated proof support like

```
sint x = 17;
sint a = 42;
sint b = -5;
sint c = 87;
sint p =
   a*(x*x) + b*x + c;
sint sign = 0;
int output;
if (p < 0) sign = -1;
if (p > 0) sign = 1;
output = open(sign|p);
```

Ideal version

```
sint x = 17;
sint a = 42;
sint b = -5;
sint c = 87;
int p =
   open(a*(x*x) + b*x + c|a,b,c);
int sign = 0;
int output;
if (p < 0) sign = -1;
if (p > 0) sign = 1;
output = sign;
```

Pragmatic version

Figure 1.20: The ideal and pragmatic versions of the polynomial program

our simple annotation language. It should also be noted that there are still many opportunities for developing optimizations at the SMCL level and optimizing SMCR. In the next section we will explore a language level optimization for reducing the number of multiplications.

```
int x = 17;
int a = 42;
int b = -5;
int c = 87;
int p = a*(x*x) + b*x + c;
int sign = 0;
int output;
if (p < 0) sign = -1;
if (p > 0) sign = 1;
output = sign;
```

Public version

Figure 1.21: The public version polynomial program

| | ideal | pragmatic | public |
|---|---|---|---|
| (3,1) | 12 sec | 30 ms | <1 ms |
| (5,2) | 17 sec | 65 ms | <1 ms |
| (7,3) | 30 sec | 132 ms | <1 ms |

Figure 1.22: The timing result in SMCR of the three versions of the polynomial program

## 1.8 Optimization

Secure multiparty computation enables computation on secret values. Secret values may be added, subtracted, multiplied, and compared like ordinary integers in some sub-field of $\mathbb{Z}$, usually $\mathbb{Z}_{32}$. However, compared to non-secret computations there is one huge difference, *speed*. The time it takes to perform a algebraic operation on secret values is large compared to performing the same operation on public values, making SMC an expensive choice for a substantial set of problems.

One possible strategy to combat the large execution time is to lower the time it takes to perform each operation. This strategy is the topic of much research in the cryptographic community [17,35,51,96]. Another orthogonal strategy is to explore the fact that some operations on secret values are substantially faster to compute than others. E.g. in SMCR it takes 1 millisecond to perform one multiplication whereas an addition can be done as fast as on non-secret values. Thus it is approx. a factor of 1000 times slower to compute a multiplication of two secret numbers than adding the same two numbers.

This observation lead us to look for ways to rewrite algebraic expressions which preserve the result of a computation but reduce the execution time. As a motivating example we have looked at the following expression over the variables $x, y, z$ containing secret values:

$$(x * y) + ((1 - x) * z)$$

another expression which evaluates to the same result, but is faster to compute is:

$$(x * (y - z)) + z$$

here we have eliminated one multiplication and kept the number of other operations constant, which made the execution time considerable faster.

What enabled us to save one multiplication was the rewriting of the expression into a variant $((x * y) - (x * z)) + z$ where we could apply the distributive law, to move the common factor $x$ outside the subtraction. However, given an expression it is not obvious to a man or machine where the distribute law can be applied or which sequence of applications results in the expression with the least number of multiplications and thus the fastest execution time.

This leaves us with the interesting question of, how to generate different variants of an expression where we can apply the distributive law. Given an expression we propose the following solution:

1. Generate all variants of the expression under the associative, commutative, and distributive laws.
2. Define and apply a cost function which counts the number of multiplications.
3. Choose the variant with the lowest cost according to the cost function.

This straight-forward and brute-force solution might seem inefficient at first. It does have an exponential worst case complexity, but the number of variables and constants in expressions found in programs are usually quite small. Our hope is that by applying a number of heuristics the algorithm turn out to be a pragmatic tool, which results in faster programs.

We will mainly focus on the basic algorithm, and prove that all the variants of expressions we generate are valid variants under the three laws. In the end of this section we will briefly look at some simple heuristics which may improve the performance, but there is still room for future work on better heuristics.

To prove the correctness of the generated variants, we define the three laws on *binary algebraic expressions*. The intention is that binary algebraic expressions will

correspond to mathematical expressions. We then define *rooted binary trees* which corresponds to the representation of expressions in the SMCL compiler and define transformations on trees corresponding to each of the three laws. We also define a mapping between binary algebraic expressions and rooted binary trees. We will show that each of the binary algebraic expressions generated by applications of the three laws corresponds uniquely to a rooted binary tree generated by applications of the tree transformations corresponding the three laws.

In the next Subsection we formally define binary algebraic expressions, rooted binary trees, and a mapping between binary algebraic expressions and rooted binary trees. These general definitions will be the foundation of the Subsections 1.8.2, 1.8.3, and 1.8.4 where we will formally define the associative, distributive, and commutative laws on binary algebraic expressions and define transformations of rooted binary trees for each of the three laws. In Subsection 1.8.5 we present the proof. In Subsection 1.8.6 we discuss and describe some heuristics.

### 1.8.1 General Definitions

We start by formally defining *binary algebraic expressions*, *rooted binary trees*, and a mapping from binary algebraic expressions to rooted binary trees. We also define some auxiliary concepts like *fixpoint* and *sub-expression/sub-tree* substitution and prove some lemmas, which will be useful in later subsections.

A *binary algebraic expression* is a string of balanced parenthesis and constants *consts* (integer literals and variables), composed using binary operators *binops*.

**Definition 1.14** *A* binary algebraic expression $e$ *in the set of binary algebraic expressions* $\mathrm{E}$ *over a set of constants consts and a set of binary operators binops is any string constructed as follows*

1. $a \in consts$ *is a binary algebraic expression*
2. *If* $\circ \in binops$*, and* $e_1, e_2$ *are binary algebraic expressions then* $(e_1) \circ (e_2)$ *is a binary algebraic expression.*

From now on we will refer to binary algebraic expressions as just expressions.

We define a rooted binary tree as an inductive structure of either a leaf $a \in consts$ or a root consisting of two rooted binary trees $t_1, t_2$ and a binary operator $\circ$.

**Definition 1.15** *A* rooted binary tree $t$ *in the set of rooted binary trees* $\mathrm{T}$ *is constructed as follows:*

1. *A leaf,* $(a)$ *where* $a \in consts$
2. *A root,* $(t_1, \circ, t_2)$ *where* $t_1, t_2 \in \mathrm{T}$ *and* $\circ \in binops$

From now on we will refer to rooted binary trees as just trees.

An expression $e$ can be encoded as a tree. We write the *tree encoding* of an expression $e$ as $[\![e]\!]$.

**Definition 1.16** *The* tree encoding $[\![e]\!]$ *of an expression* $e$ *is defined by the function:*

1. *If* $e = a \in consts$ *then* $[\![e]\!] = (a)$
2. *If* $e = (e_1) \circ_3 (e_2)$*,* $[\![e_1]\!] = (t_1, \circ_1, t_2)$*, and* $[\![e_2]\!] = (t_1', \circ_2, t_2')$ *then we define the tree encoding of* $e$ *as* $[\![e]\!] = ((t_1, \circ_1, t_2), \circ_3, (t_1', \circ_2, t_2'))$ $\forall i \in [1, 3] \ : \ \circ_i \in binops$

We will need the fact that the tree encoding has an inverse, called the *inverse tree encoding* written $[\![e]\!]^{-1}$ where $[\![e]\!]^{-1} = e$, given an expression $e$. The inverse tree encoding is well-defined if the tree encoding is bijective.

**Lemma 1.3** *The tree encoding* $[\![e]\!] : \mathrm{E} \to \mathrm{T}$ *is a* bijective *function from the set of expressions to the set of trees.*

*Proof.* By induction in the structure of the tree. □

Here $\mathrm{E} \to \mathrm{T}$ denote the type of the tree encoding function. The type is given as aid for the reader.

We generate all variants under each of the three laws by iteratively applying the law or corresponding tree transformation. The process is guaranteed to reach a fixpoint because expressions and trees are finite. To compute the fixpoint of a function we define the *fixpoint* function $\mathrm{fix} : (\alpha \to \alpha) \to \alpha$. Where $(\alpha \to \alpha) \to \alpha$ is the type of the function and $\alpha$ is a type variable.

**Definition 1.17** *The* fixpoint *function* $\mathrm{fix} : (\alpha \to \alpha) \to \alpha$ *computes the least fixpoint of a function* $f : \alpha \to \alpha$*, and has the following property:*

$$f(\mathrm{fix}(f)) = \mathrm{fix}(f)$$

To define sub-expression/sub-tree substitution, we recursively define the sets of sub-expressions $\Gamma(e)$ and sub-trees $\Psi(t)$ for expression $e$ and tree $t$ respectively by decomposing the expression or tree and collect all sub-expressions or sub-trees.

**Definition 1.18** *The set of* sub-expressions $\Gamma(e)$ *of an expression* $e$ *is defined as:*

$$\Gamma(e) = \begin{cases} \{e_1, e_2\} \cup \Gamma(e_1) \cup \Gamma(e_2) & \text{if } e = (e_1) \circ (e_2) \\ \emptyset & \text{if } e \in \mathit{consts} \end{cases}$$

**Definition 1.19** *The set of* sub-trees $\Psi(t)$ *of a tree* $t$ *is defined as:*

$$\Psi(t) = \begin{cases} \{t\} \cup \Psi(t_1) \cup \Psi(t_2) & \text{if } t = (t_1, \circ, t_2) \\ \{t\} & \text{if } t = (a) \end{cases}$$

Sub-expression and sub-tree substitution are recursively defined in terms of the sets of sub-expressions and sub-trees respectively. The substitution recursively decompose the expression or tree until the sub-expressions or sub-tree to be substituted is located and then perform the substitution.

**Definition 1.20** Sub-expression substitution *on an expression* $e$ *written* $e\big[\gamma \to e'\big]$*, where* $\gamma \in \Gamma(e)$ *and* $e'$ *is an expression, is defined as:*

$$e\big[\gamma \to e'\big] = \begin{cases} e' & \text{if } e = \gamma \\ e_1\big[\gamma \to e'\big] \circ e_2\big[\gamma \to e'\big] & \text{if } e \neq \gamma \wedge e = (e_1) \circ (e_2) \\ e & \text{if } e \neq \gamma \wedge e \in \mathit{consts} \end{cases}$$

**Definition 1.21** Sub-Tree substitution *on trees written* $t[\psi \to t']$*, where* $\psi \in \Psi(t)$ *and* $t'$ *is a tree, is defined as:*

$$t\big[\psi \to t'\big] = \begin{cases} t' & \text{if } t = \psi \\ (t_1\big[\psi \to t'\big], \circ, t_2\big[\psi \to t'\big]) & \text{if } t \neq \psi \wedge t = (t_1, \circ, t_2)) \\ t & \text{if } t \neq \psi \wedge t = (a) \end{cases}$$

The following lemma intuitively says that substitution composes with tree encoding.

**Lemma 1.4** *If* $e, e'$ *are expressions and* $\gamma \in \Gamma(e)$ *is a sub-expression in* $e$*,* $\psi \in \Psi(t)$ *is a sub-tree in* $t$*, where* $t = [\![e]\!]$*,* $[\![\gamma]\!] = \psi$*, and* $t' = [\![e']\!]$ *then* $[\![e\big[\gamma \to e'\big]]\!] = [\![e]\!]\big[\psi \to t'\big]$*.*

*Proof.* Proof by induction in the structure of the of $e$ □

### 1.8.2 Definition of Associativity

First we give a general definition of associativity for a binary operator $\circ$ on a set $S$, and then we define associativity for expressions and trees.

A binary operator $\circ$ on a set $S$ is called associative if it satisfies the associative law:

$$((x) \circ (y)) \circ (z) = (x) \circ ((y) \circ (z)) \ \forall x, y, z \in S$$

Associativity for expressions is defined similarly. We define the associative expression transformation $f_{\mathrm{a}}$ as a function which takes an expression from the set of all expressions E and returns an expression also in E.

**Definition 1.22** *The* associative expression transformation $f_{\mathrm{a}} : \mathrm{E} \to \mathrm{E}$ *is defined as follows:*

$$f_{\mathrm{a}}(e) = \begin{cases} (e_1) \circ ((e_2) \circ (e_3)) & \text{if } e = ((e_1) \circ (e_2)) \circ (e_3) \\ ((e_1) \circ (e_2)) \circ (e_3) & \text{if } e = (e_1) \circ ((e_2) \circ (e_3)) \\ e & \text{else} \end{cases}$$

We note that $f_{\mathrm{a}}$ is its own inverse $f_{\mathrm{a}} = f_{\mathrm{a}}^{-1}$.

Similarly we define associativity for trees by the associative tree transformation $\Delta_{\mathrm{a}}$ as a function which takes a tree from the set T of all trees and returns a tree also in T.

**Definition 1.23** *The* associative tree transformation $\Delta_{\mathrm{a}} : \mathrm{T} \to \mathrm{T}$ *is defined as follows:*

$$\Delta_{\mathrm{a}}(t) = \begin{cases} (a, \circ, (b, \circ, c)) & \text{if } t = ((a, \circ, b), \circ, c) \\ ((a, \circ, b), \circ, c) & \text{if } t = (a, \circ, (b, \circ, c)) \\ t & \text{else} \end{cases}$$

We note that $\Delta_{\mathrm{a}}$ is its own inverse $\Delta_{\mathrm{a}} = \Delta_{\mathrm{a}}^{-1}$.

We define the associative expression relation as a relation on expressions, which makes two expressions equivalent if one has a sub-expression which can be rewritten using the associative expression transformation to obtain the other.

**Definition 1.24** *The* associative expression relation $\sim_{\mathrm{a}} \subseteq \mathrm{E} \times \mathrm{E}$ *is a relation on expressions. We say that* $e$ *and* $e'$ *are* associative expression equivalent *written* $e \sim_{\mathrm{a}} e'$*, if there is a* $\gamma \in \Gamma(e)$ *such that* $e' = e\big[\gamma \to f_{\mathrm{a}}(\gamma)\big]$.

Similarly we define the associative tree relation as a relation on trees, which makes two trees equivalent if one has a sub-tree which can be rewritten using the associative tree transformation to obtain the other.

**Definition 1.25** *The* associative tree relation*, written* $\sim_{\Delta_{\mathrm{a}}} \subseteq \mathrm{T} \times \mathrm{T}$*, is a relation defined on trees* $t, t'$*. We say that* $t$ *and* $t'$ *are* associative tree equivalent *if there is a* $\psi \in \Psi(t)$ *and* $t' = t\big[\psi \to \Delta_{\mathrm{a}}\psi\big]$.

We define the associative transitive expression relation as the transitive extension of the associative expression relation.

**Definition 1.26** *Expressions* $e, e'$ *are in the associative* transitive *expression relation* $\sim_{\mathrm{a}}^{*} \subseteq \mathrm{E} \times \mathrm{E}$ *written* $e \sim_{\mathrm{a}}^{*} e'$*, if there exists* $e_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$e \sim_{\mathrm{a}} e_1 \sim_{\mathrm{a}} e_2 \sim_{\mathrm{a}} \ldots \sim_{\mathrm{a}} e_n \sim_{\mathrm{a}} e'$$

*We say that* $e$ *and* $e'$ *are* associative transitive expression equivalent *if* $e \sim_{\mathrm{a}}^{*} e'$.

Similarly we define the associative transitive tree relation as the transitive extension of the associative tree relation.

**Definition 1.27** *Trees $t, t'$ are in the* transitive *associative relation* $\sim^*_{\Delta_a} \subseteq \mathrm{T} \times \mathrm{T}$*, written* $t \sim^*_{\Delta_a} t'$*, if there exists* $t_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$t \sim_{\Delta_a} t_1 \sim_{\Delta a} t_2 \sim_{\Delta_a} \ldots \sim_{\Delta_a} t_n \sim_{\Delta_a} t'$$

*We say that* $t$ *and* $t'$ *are* associative transitive tree equivalent *if* $t \sim^*_{\Delta_a} t'$*.*

We will need the fact that the associative transitive expression relation is an equivalence relation, because we will in effect be computing the equivalence class of this relation.

**Lemma 1.5** *The associative transitive expression relation is an equivalence relation.*

Similarly we will be computing the equivalence class of the associative transitive tree relation, and so we need to know that the relation is an equivalence relation.

**Lemma 1.6** *The associative transitive tree relation is an equivalence relation.*

### 1.8.3 Definition of Commutativity

First we give a general definition of commutativity for a binary operator $\circ$ on a set $S$, and then we define commutativity for expressions and trees.

A binary operator $\circ$ on a set S is called a commutative operator if it satisfies the commutative law:

$$(x) \circ (y) = (y) \circ (x) \; \forall x, y \in S$$

Commutativity for expressions is defined similarly. We define the commutative expression transformation $f_c$ as a function which takes an expression from the set of all expressions E and returns an expression also in E.

**Definition 1.28** *The* commutative expression transformation $f_c : \mathrm{E} \to \mathrm{E}$ *is defined as follows:*

$$f_c(e) = \begin{cases} (e_1) \circ (e_2) & \text{if } e = (e_2) \circ (e_1) \\ (e_2) \circ (e_1) & \text{if } e = (e_2) \circ (e_1) \\ e & \text{else} \end{cases}$$

We note that $f_c$ is its own inverse $f_c = f_c^{-1}$.

Similarly we define commutativity for trees by the commutative tree transformation $\Delta_c$ as a function which takes a tree from the set T of all trees and returns a tree also in T.

**Definition 1.29** *The* commutative tree transformation $\Delta_c : \mathrm{T} \to \mathrm{T}$ *is defined as follows:*

$$\Delta_c(t) = \begin{cases} (a, \circ, b) & \text{if } t = (b, \circ, a) \\ (b \circ, a) & \text{if } t = (a, \circ, b) \\ t & \text{else} \end{cases}$$

We note that $\Delta_c$ is its own inverse $\Delta_c = \Delta_c^{-1}$.

We define the commutative expression relation as a relation on expressions, which makes two expressions equivalent if one has a sub-expression which can be rewritten using the commutative expression transformation to obtain the other.

**Definition 1.30** *The* commutative expression relation $\sim_c \subseteq \mathrm{E} \times \mathrm{E}$ *is a relation on expressions. We say that* $e$ *and* $e'$ *are* commutative expression equivalent *written* $e \sim_c e'$*, if there is a* $\gamma \in \Gamma(e)$ *such that* $e' = e\big[\gamma \to f_c(\gamma)\big]$*.*

Similarly we define the commutative tree relation as a relation on trees, which makes two trees equivalent if one has a sub-tree which can be rewritten using the commutative tree transformation to obtain the other.

**Definition 1.31** *The* commutative tree relation*, written* $\sim_{\Delta_c} \subseteq \mathrm{T} \times \mathrm{T}$*, is a relation defined on trees* $t, t'$*. We say that* $t$ *and* $t'$ *are* commutative tree equivalent *if there is a* $\psi \in \Psi(t)$ *and* $t' = t\big[\psi \to \Delta_c \psi\big]$*.*

We define the commutative transitive expression relation as the transitive extension of the commutative expression relation.

**Definition 1.32** *Expressions* $e, e'$ *are in the commutative* transitive *expression relation* $\sim_c^* \subseteq \mathrm{E} \times \mathrm{E}$ *written* $e \sim_c^* e'$*, if there exists* $e_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$e \sim_c e_1 \sim_c e_2 \sim_c \ldots \sim_c e_n \sim_c e'$$

*We say that* $e$ *and* $e'$ *are* commutative transitive expression equivalent *if* $e \sim_c^* e'$*.*

Similarly we define the commutative transitive tree relation as the transitive extension of the commutative tree relation.

**Definition 1.33** *Trees* $t, t'$ *are in the* transitive *commutative relation* $\sim_{\Delta_c}^* \subseteq \mathrm{T} \times \mathrm{T}$*, written* $t \sim_{\Delta_c}^* t'$*, if there exists* $t_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$t \sim_{\Delta_c} t_1 \sim_{\Delta c} t_2 \sim_{\Delta_c} \ldots \sim_{\Delta_c} t_n \sim_{\Delta_c} t'$$

*We say that* $t$ *and* $t'$ *are* commutative transitive tree equivalent *if* $t \sim_{\Delta_c}^* t'$*.*

We will need the fact that the commutative transitive expression relation is an equivalence relation, because we will in effect be computing the equivalence class of this relation.

**Lemma 1.7** *The commutative transitive relation is an equivalence relation.*

Similarly we will be computing the equivalence class of the commutative transitive tree relation, and so we need to know that the relation is an equivalence relation.

**Lemma 1.8** *The commutative transitive tree relation is an equivalence relation.*

### 1.8.4 Definition of Distributivity

The distributive law can be decomposed into two, the left-distributive and the right-distributive laws, which have the nice properties that they are one-to-one and their own inverse. We therefore start with a general definition of left-distributivity and right-distributivity for two binary operators $\circ_1$ and $\circ_2$ on a set $S$, and then we define left-distributivity and right-distributivity for expressions and trees.

Given two binary operators $\circ_1$ and $\circ_2$ on a set S, we say that the operator $\circ_1$

- is left-distributive over $\circ_2$ if, given any elements $x$, $y$, and $z \in S$:

$$x \circ_1 (y \circ_2 z) = (x \circ_1 y) \circ_2 (x \circ_1 z)$$

- is right-distributive over $\circ_2$ if, given any elements $x$, $y$, and $z \in S$:

$$(y \circ_2 z) \circ_1 x = (y \circ_1 x) \circ_2 (z \circ_1 x)$$

Left- and right-distributivity for expressions is defined similarly. We define the left- and right-distributive expression transformations $f_{\mathrm{ld}}$ and $f_{\mathrm{rd}}$ as functions which takes an expression from the set of all expressions E and returns an expression also in E, respectively.

**Definition 1.34** *The* left-distributive expression transformation $f_{ld} : \mathrm{E} \to \mathrm{E}$ *is defined as follows:*

$$f_{\mathrm{ld}}(e) = \begin{cases} (e_1) \circ_1 ((e_2) \circ_2 (e_3)) & \text{if } e = ((e_1) \circ_1 (e_2)) \circ_2 ((e_1) \circ_1 (e_3)) \\ ((e_1) \circ_1 (e_2)) \circ_2 ((e_1) \circ_1 (e_3)) & \text{if } e = (e_1) \circ_1 ((e_2) \circ_2 (e_3)) \\ e & \text{else} \end{cases}$$

**Definition 1.35** *The* right-distributive expression transformation $f_{\mathrm{rd}} : \mathrm{E} \to \mathrm{E}$ *is defined as follows:*

$$f_{\mathrm{rd}}(e) = \begin{cases} ((e_2) \circ_2 (e_3)) \circ_1 (e_1) & \text{if } e = ((e_2) \circ_1 (e_1)) \circ_2 ((e_3) \circ_1 (e_1)) \\ ((e_2) \circ_1 (e_1)) \circ_2 ((e_3) \circ_1 (e_1)) & \text{if } e = ((e_2) \circ_2 (e_3)) \circ_1 (e_1) \\ e & \text{else} \end{cases}$$

We note that $f_{\mathrm{ld}}$ and $f_{\mathrm{rd}}$ are their own inverse $f_{\mathrm{ld}} = f_{\mathrm{ld}}^{-1}$ and $f_{\mathrm{rd}} = f_{\mathrm{rd}}^{-1}$.

Similarly we define left- and right-distributivity for trees by the left- and right- tree transformations $\Delta_{\mathrm{ld}}$ and $\Delta_{\mathrm{rd}}$ as functions which takes a tree from the set T of all trees and returns a tree also in T, respectively.

**Definition 1.36** *The* left distributive tree transformation $\Delta_{ld} : \mathrm{T} \to \mathrm{T}$ *is defined as follows:*

$$\Delta_{ld}(t) = \begin{cases} (a, \circ_1, (b, \circ_2, c)) & \text{if } t = ((a, \circ_1, b), \circ_2, (a, \circ_1, c)) \\ ((a, \circ_1, b), \circ_2, (a, \circ_1, c)) & \text{if } t = (a, \circ_1, (b, \circ_2, c)) \\ t & \text{else} \end{cases}$$

**Definition 1.37** *The* right distributive tree transformation $\Delta_{rd} : \mathrm{T} \to \mathrm{T}$ *is defined as follows:*

$$\Delta_{rd}(t) = \begin{cases} ((b, \circ_2, c), \circ_1, a) & \text{if } t = ((b, \circ_1, a), \circ_2, (c, \circ_1, a))) \\ ((b, \circ_1, a), \circ_2, (c, \circ_1, a)) & \text{if } t = ((b, \circ_2, c), \circ_1, a) \\ t & \text{else} \end{cases}$$

We also note that $\Delta_{\mathrm{ld}}$ and $\Delta_{\mathrm{rd}}$ are their own inverse $\Delta_{\mathrm{ld}} = \Delta_{\mathrm{ld}}^{-1}$ and $\Delta_{\mathrm{rd}} = \Delta_{\mathrm{rd}}^{-1}$.

We define the left- and right-distributive expression relation as a relation on expressions, which makes two expressions equivalent if one has a sub-expression which can be rewritten using the left- or right-distributive expression transformation, respectively to obtain the other.

**Definition 1.38** *The* left-distributive expression relation $\sim_{\mathrm{ld}} \subseteq \mathrm{E} \times \mathrm{E}$ *is a relation on expressions. We say that* $e$ *and* $e'$ *are* left-distributive expression equivalent *written* $e \sim_{\mathrm{ld}} e'$*, if there is a* $\gamma \in \Gamma(e)$ *such that* $e' = e\big[\gamma \to f_{\mathrm{ld}}(\gamma)\big]$.

**Definition 1.39** *The* right-distributive expression relation $\sim_{\mathrm{rd}} \subseteq \mathrm{E} \times \mathrm{E}$ *is a relation on expressions. We say that* $e$ *and* $e'$ *are* right-distributive expression equivalent *written* $e \sim_{\mathrm{rd}} e'$*, if there is a* $\gamma \in \Gamma(e)$ *such that* $e' = e\big[\gamma \to f_{\mathrm{rd}}(\gamma)\big]$.

Similarly we define the left- and right-distributive tree relation as a relation on trees, which makes two trees equivalent if one has a sub-tree which can be rewritten using the left- or right-distributive tree transformation to obtain the other.

**Definition 1.40** *The* left-distributive tree relation $\sim_{\Delta_{\mathrm{ld}}} \subseteq \mathrm{E} \times \mathrm{E}$ *is a relation on trees. We say that* $t$ *and* $t'$ *are* left-distributive tree equivalent *written* $t \sim_{\Delta_{\mathrm{ld}}} t'$*, if there is a* $\psi \in \Psi(t)$ *such that* $t' = t\big[\psi \to \Delta_{\mathrm{ld}}(\psi)\big]$.

**Definition 1.41** *The* right-distributive tree relation $\sim_{\Delta_{\mathrm{rd}}} \subseteq \mathrm{E} \times \mathrm{E}$ *is a relation on trees. We say that* $t$ *and* $t'$ *are* right-distributive tree equivalent *written* $t \sim_{\Delta_{\mathrm{rd}}} t'$*, if there is a* $\psi \in \Psi(t)$ *such that* $t' = t\big[\psi \to \Delta_{\mathrm{rd}}(\psi)\big]$.

We define the left- and right-distributive transitive expression relation as the transitive extension of the left- and right-distributive expression relation, respectively.

**Definition 1.42** *Expressions* $e, e'$ *are in the left-distributive* transitive *expression relation* $\sim^*_{\mathrm{ld}} \subseteq \mathrm{E} \times \mathrm{E}$ *written* $e \sim^*_{\mathrm{ld}} e'$*, if there exists* $e_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$e \sim_{\mathrm{ld}} e_1 \sim_{\mathrm{ld}} e_2 \sim_{\mathrm{ld}} \ldots \sim_{\mathrm{ld}} e_n \sim_{\mathrm{ld}} e'$$

*We say that* $e$ *and* $e'$ *are* left-distributive transitive expression equivalent *if* $e \sim^*_{\mathrm{ld}} e'$.

**Definition 1.43** *Expressions* $e, e'$ *are in the right-distributive* transitive *expression relation* $\sim^*_{\mathrm{rd}} \subseteq \mathrm{E} \times \mathrm{E}$ *written* $e \sim^*_{\mathrm{rd}} e'$*, if there exists* $e_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$e \sim_{\mathrm{rd}} e_1 \sim_{\mathrm{rd}} e_2 \sim_{\mathrm{rd}} \ldots \sim_{\mathrm{rd}} e_n \sim_{\mathrm{rd}} e'$$

*We say that* $e$ *and* $e'$ *are* rd-distributive transitive expression equivalent *if* $e \sim^*_{\mathrm{rd}} e'$.

Similarly we define the left- and right-distributive transitive tree relation as the transitive extension of the left- and right-distributive tree relation, respectively.

**Definition 1.44** *Trees* $t, t'$ *are in the left-distributive* transitive *relation* $\sim^*_{\Delta_{\mathrm{ld}}} \subseteq \mathrm{T} \times \mathrm{T}$*, written* $t \sim^*_{\Delta_{\mathrm{ld}}} t'$*, if there exists* $t_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$t \sim_{\Delta_{\mathrm{ld}}} t_1 \sim_{\Delta_{\mathrm{ld}}} t_2 \sim_{\Delta_{\mathrm{ld}}} \ldots \sim_{\Delta_{\mathrm{ld}}} t_n \sim_{\Delta_{\mathrm{ld}}} t'$$

*We say that* $t$ *and* $t'$ *are* left-distributive transitive tree equivalent *if* $t \sim^*_{\Delta_{\mathrm{ld}}} t'$.

**Definition 1.45** *Trees* $t, t'$ *are in the right-distributive* transitive *relation* $\sim^*_{\Delta_{\mathrm{rd}}} \subseteq \mathrm{T} \times \mathrm{T}$*, written* $t \sim^*_{\Delta_{\mathrm{rd}}} t'$*, if there exists* $t_i$ *for* $i \in [1, n]$ *for some* $n$ *such that:*

$$t \sim_{\Delta_{\mathrm{rd}}} t_1 \sim_{\Delta_{\mathrm{rd}}} t_2 \sim_{\Delta_{\mathrm{rd}}} \ldots \sim_{\Delta_{\mathrm{rd}}} t_n \sim_{\Delta_{\mathrm{rd}}} t'$$

*We say that* $t$ *and* $t'$ *are* right-distributive transitive tree equivalent *if* $t \sim^*_{\Delta_{\mathrm{rd}}} t'$.

We will need the fact that the left- and right-distributive transitive expression relations are equivalence relations, because we will in effect be computing the equivalence classes of the relations.

**Lemma 1.9** *The left- and right-distributive transitive relations are equivalence relations.*

Similarly we will be computing the equivalence classes of the left- and right-distributive transitive tree relations, and so we need to know that the relations are equivalence relations.

**Lemma 1.10** *The left- and right-distributive transitive tree relations are equivalence relations.*

### 1.8.5 Correctness

We prove that all trees generated using the tree transformation defined for each of the associative, commutative, and distributive laws will correspond uniquely to the expressions generated using the expression transformation for the three laws, respectively.

The proof is similar for all three laws, except that for the distributive law we prove it separately for left-distributivity and right-distributivity. We introduce the variable l to range over the three laws. We will use it as subscript on the various concepts defined above, thus $\mathrm{l} \in \{\mathrm{a}, \mathrm{c}, \mathrm{ld}, \mathrm{rd}\}$.

We want to prove a property of iterative application of the respective laws. We build up the proof in a number of steps, by relating a number of concepts under the tree encoding. First we relate the expression transformation and the tree transformation in Lemma 1.11, then we relate the expression relation and the tree relation in Lemma 1.12, and we relate the transitive expression relation with the transitive tree relation in Lemma 1.13. Finally we introduce the expression equivalence generator and the tree equivalence generator, and relate the fixpoints of these, also under the tree encoding in Theorem 1.5.

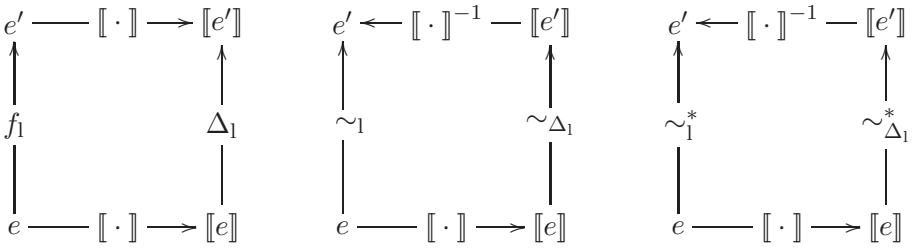



Figure 1.23: (Left) Applying the expression transformation and then the tree encoding gives the same result as applying the tree encoding and then the tree transformation. (Middel) If $e, e'$ are expression equivalent then the corresponding trees are tree equivalent. (Right) If $e, e'$ are transitive expression equivalent then the corresponding trees are transitive tree equivalent

Figure 1.23 shows the initial three properties as commutative diagrams. We begin with the left diagram, which informally states that applying the expression transformation and then the tree encoding gives the same result as applying the tree encoding and then the tree transformation.

**Lemma 1.11** *Let $e$ be an expression, then $\Delta_\mathrm{l}[\![e]\!] = [\![f_\mathrm{l}(e)]\!]$.*

*Proof.* The proof is by structural induction on $e$, where we apply the definition of $\Delta_\mathrm{l}$ and $f_\mathrm{l}$. □

The middel diagram informally says that if an expression and a tree are related by the tree encoding then so are the equivalent expression and tree.

**Lemma 1.12** *If $e, e'$ are expressions and $e \sim_\mathrm{l} e'$, $[\![e]\!] \sim_{\Delta_\mathrm{l}} [\![e'']\!]$, and $\psi = [\![\gamma]\!]$, where $\gamma$ and $\psi$ are the sub-expression and sub-tree where the transformations are applied, respectively, then $e' = [\![e'']\!]^{-1}$*

*Proof.* The proof uses the definition of $\sim_\mathrm{l}$, substitution, Lemma 1.4, and the inverse tree transformation. □

The diagram on the right of Figure 1.23 is the transitive extension of the middle diagram. Here we apply the transitive equivalences $\sim^*_\mathrm{l}$ and $\sim^*_{\Delta_\mathrm{l}}$ on expressions and

trees and show that if the starting expression and tree are related by the tree encoding then so are the results.

**Lemma 1.13** *If $e, e'$ are expressions and $e \sim_{\mathrm{l}}^{*} e'$ then $[\![e]\!] \sim_{\Delta_{\mathrm{l}}}^{*} [\![e']\!]$.*

*Proof.* The proof proceeds by induction on the number of transformations and uses Lemma 1.12. □

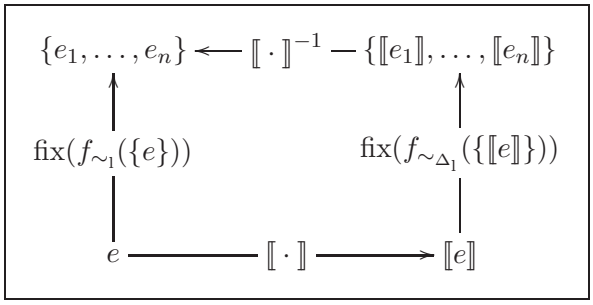



Figure 1.24: Applying the tree encoding to the expression $e$ and then computing the fixpoint of the tree equivalence generator gives a set of trees that are related by the inverse tree transformation to the set of expressions computed by the fixpoint of the expression equivalence generator applied to $e$.

We define the *expression equivalence generator* as a function $f_{\sim_{\mathrm{l}}} : 2^{\mathrm{E}} \to 2^{\mathrm{E}}$ which takes a set of expressions as argument and computes the set of expressions which are equivalent to each of the expressions in the argument under the expression relation $\sim_{\mathrm{l}}$.

**Definition 1.46** *The expression equivalence generator is a function $f_{\sim_{\mathrm{l}}}$ defined as:*

$$f_{\sim_{\mathrm{l}}}(X) = \cup_{\forall e \in X} \{e' \mid e' \sim_{\mathrm{l}} e\}$$

We define the *tree equivalence generator* as a function $f_{\Delta_{\mathrm{l}}} : 2^{\mathrm{T}} \to 2^{\mathrm{T}}$ which takes a set of trees as argument and computes the set of trees which are equivalent to each of the trees in the argument under the tree relation $\sim_{\Delta_{\mathrm{l}}}$.

**Definition 1.47** *The tree equivalence generator is a function $f_{\Delta_{\mathrm{l}}}$ defined as:*

$$f_{\Delta_{\mathrm{l}}}(X) = \cup_{\forall t \in X} \{t' \mid t' \sim_{\Delta_{\mathrm{l}}} t\}$$

All the expressions generated by computing the fixpoint of the expression equivalence generator are transitive expression equivalent to the element in the argument set. Similar for the fix point of the tree equivalence generator.

**Lemma 1.14** $\mathrm{fix}(f_{\sim_{\mathrm{l}}}(\{e\})) = \{e_1, \ldots, e_n\} \Rightarrow e \sim_{\mathrm{l}}^{*} e_i$ *where* $i \in [1, \ldots, n]$. *Similar for* $\mathrm{fix}(f_{\Delta_{\mathrm{l}}}(\{[\![e]\!]\})) = \{[\![e_1]\!], \ldots, [\![e_n]\!]\} \Rightarrow [\![e]\!] \sim_{\Delta_{\mathrm{l}}}^{*} [\![e_i]\!]$ .

*Proof.* By induction in the number of fixpoint iterations, which is finite, because the number of expressions equivalent to a given expression is finite. □

Lemma 1.14 does most of the work for Theorem 1.5, which proves that each of the trees in the set computed by the fixpoint of the tree equivalence generator is the tree encoding of an expression in the set computed by the fixpoint of the expression equivalence generator, and vice versa.

**Theorem 1.5** *If* $\text{fix}(f_{\sim_l}(\{e\})) = \{e_1, \ldots, e_n\}$ *where* $e$ *and* $e_i$ *for* $i \in [1, n]$ *are expressions then* $\text{fix}(f_{\Delta_l}(\{[\![e]\!]\})) = \{[\![e_1]\!], \ldots, [\![e_n]\!]\}$.

*Proof.* Assume for contradiction that the above is not true. Let $A = \text{fix}(f_{\sim_l}(\{e\}))$ and $B = \text{fix}(f_{\Delta_l}(\{[\![e]\!]\}))$. Then either:

i $\exists e'$ s.t. $e' \in A \wedge [\![e']\!] \notin B$

ii $\exists e'$ s.t. $e' \notin A \wedge [\![e']\!] \in B$

Case i. We assume that $\exists e'$ s.t. $e' \in A \wedge [\![e']\!] \notin B$. According to Lemma 1.14 we know that $e \sim_l^* e'$, and according to Lemma 1.13 there exists a sequence of transformations such that $[\![e]\!] \sim_{\Delta_l}^* [\![e']\!]$. By the definition of fix we see that $[\![e']\!] \in B$, which contradicts the assumption that $[\![e']\!] \notin B$.

Case ii. We assume that $\exists e'$ s.t. $e' \notin A \wedge [\![e']\!] \in B$. According to Lemma 1.14 we know that $\exists e$ s.t. $[\![e]\!] \sim_{\Delta_l}^* [\![e']\!]$, now by Lemma 1.13 there exists a sequence of transformations such that $e \sim_l^* e'$. By the definition of fix we see that $e' \in A$, which contradicts the assumption that $e' \notin B$. □

We now conclude that all the trees generated by computing the fixpoint of the tree equivalence generator and thus the tree transformation does indeed correspond to all the expressions generated by iteratively applying one of the three laws. All combinations of expressions generated under all three laws can now be generated by computing the fixpoint of the union of the individual fixpoints i.e. $\text{fix}(\bigcup_l \text{fix}(f_{\sim_l}(\{e\})))$.

### 1.8.6 Heuristics

The algorithm described above is worst-case exponential in the number of operators. Fortunately large expressions are rare in practise, because the tend to make code less readable and we are only interested in a limited number expressions, those containing multiplications. The last part gives us a good start for defining heuristics to keep the combinatorial number of expressions generated in our search within the limits of reasonable time and space.

We will use the term *sub-expression* to mean expression *or* sub-expression from on.

The algorithm is relying on the distributive law, if the distributive law cannot not apply for a sub-expression, no matter how we rewrite it, then we can treat the sub-expression as a constant. If a sub-expression does not contain multiplications then there is no need to create the combinations for that expression, and we can just treat it as a constant in the transformations defined in Subsection 1.8.5. Another point heuristic can be applied if the sub-expression does not contain the same factor in more than one place, then there is no potential for collapsing multiple occurrences.

The cost function is simple but efficient, we just count the number of multiplications in the expressions. This does not take into account how many other operations that might have been introduced by the transformations, which may be a concern since one multiplication is worth 1000 additions in terms of speed. However only the distributive transformations may potentially add more operators. If that is the case then it is because we use the distributive law to distribute a factor over some operator, which produces two more multiplications. So if this distribution does not enable some larger reduction of multiplications somewhere else in the expression, then it will be deemed more expensive by the cost function than the original expression. We believe that the cost function is simple and adequate.

## 1.9 Related Work

To the best of our knowledge, SMCL is the first imperative programming language for general Secure Multiparty Computation. We discuss its relation to two other languages for SMC, and we survey the areas of language-based information-flow security and cryptography and explain their relationship to our work.

### 1.9.1 Languages for SMC

Closely related is the Fairplay project [62], which has developed two DSLs for secure function evaluation (that is the special case of SMC where the input is only provided in the beginning), the Fairplay [62] and FairplayMP [14] systems. Both Fairplay systems consists of a compiler from a C like procedural language called the Secure Function Definition Language (SFDL) to one-pass boolean circuits described in the Secure Hardware Definition Language (SHDL). This separation of concern is similar to our approach where we compile SMCL to Java with calls to the SMCR API. The runtime for Fairplay and FairplayMP have the same level of security as the SMCR, both are immune to attacks from passive (semi-honest) attackers.

SFDL exists in a version for each of the two systems. The versions have a number of things in common: They are DSLs with a C like syntax, all values are secret boolean, integer, enumerations, or C like structs. They also support arrays and the usual logic and arithmetic operations on booleans and integers except for multiplication and division on integers.

Both versions allow specification of input and output to the parties, however in Fairplay this is restricted to two parties whereas FairplayMP allows any number of parties to participate. FairplayMP divides parties according to the role they play in the computation, in much the same way SMCL [69] does. In FairplayMP there are three different kinds of parties: The Input Parties(IP), The Computation Parties(CP), and the Output Parties(OP). IPs provides input, CP carries out the construction and evaluation of the boolean circuit, and the OPs learns a result of the evaluation. This division has the same benefits as our model of clients and a server does.

The main difference between SMCL and both versions of Fairplay is that SMCL allows both public/private and secret values which may potentially boost efficiency and allows not only secure function evaluation but general loops and recursive functions on public/private values.The Fairplay systems only allow Secure Function Evaluation. A reactive system can be built on top of Fairplay, by intergrating it with a general purpose language, however it is not obvious how that would work and Fairplay does not address the security problems that will arise from such an integration like we do with SMCL.

A minor difference is how to range over clients/IPs of the same kind. In SMCL one can define groups of clients. A group is lifted to a unique datatype in SMCL, whereas in FairplayMP it is easily realized using structs. Which to prefer is a matter of taste.

Another closely related language is the SMC language [87]. The language is a declarative language for SMC based on constraint programming. A public program is distributed among the parties in the computation along with an interpreter, each party inputs his secret values and the interpreter calculates the result. Computations are specified as arithmetic circuits and lacks branches on secret values. The computer of each party is considered secure in contrast to SMCL where the computation is done at the server parties, which we do not consider secure. SMCL is more expressive, offers stricter security guarantees, and provides a higher abstraction level.

### 1.9.2 Language-Based Security

Language based information-flow security aims at developing language mechanisms for protection against deliberate or accidental release of information. A thorough survey of language-based security is given by Sabelfeld and Myers in [78]. To SMCL the protection of confidential information is of vital importance and SMCL applies information-flow control to enforce security. Below we discuss relevant areas of language-based security.

#### Noninterference

Denning [36] was the first to present a solution to the noninterference problem in terms of a static analysis which prevents explicit as well as implicit flow of information, however Denning did not provide a formal argument of noninterference.

Volpano and Smith [103] recast the work of Denning in terms of a type system for a simple imperative language and prove that well-typed programs obey the noninterference property. The type system is based on the lattice proposed by Denning. The partial ordering on the lattice extends naturally to a subtype relation. Types are divided into security levels and variable- and command types. A variable has a type $\tau$-var if it contains values of security level $\tau$ or lower, while a command has type $\tau$-cmd if no assignments occur in the command to a variable of security level lower than $\tau$. The command-types are similar to the program counter label used by others like Sabelfeld in [110]. The work by Volpano and Smith has ignited a wide variety of work on the use of security types for enforcing noninterference. The term security types has been coined by Sabelfeld and Myers in [78] to denote annotation based approaches, e.g. type systems, to noninterference. The work has been extended in various directions to languages with first-order procedures [101], multiple threads [81, 88], and concurrent programs [102]. The type system based approach has been applied to wide range of settings like the calculi SLam [49] and DCC [2], the functional language FlowCaml [75], and recently VHDL [98].

Another approach to noninterference is the use of an "information-flow logic". Amtoft et al. propose a Hoare-like logic [9, 10] on top of which they present an interprocedural and modular information-flow analysis where noninterference is enforced as an end-to-end guarantee in object-oriented programs and programs with pointers. The logic supports programmer assertions that specify more precise information-flow policies. The technique has increased precision compared to previous type-based approaches like the ones by Volpano and Smith [101] and Zdancewic [110].

SMCL is a security-typed language which is firmly based on the work by Denning and by Volpano and Smith and is in line with the work done by others [43, 78, 98, 110]. SMCL basically employs a two-level lattice of security levels, a type system based on [101, 103] (in the current implementation), which together with a semantics where the trace is independent of secret values to enforce noninterference.

In the decentralized label model (DLM) of [66], information is marked by labels. A label is a set of components consisting of an owner section and a reader section. The purpose of labels is to protect the confidentiality of the owner principals that may grant other principals the right to read their values. The DLM guarantees that the privacy of principals is never compromised. SMCL is also concerned with protecting the privacy of client input, and it appealing and compelling that the DLM would be suitable for SMCL to guarantee that the values from some kind of clients do not flow to certain other clients. SMCL already has the notion of groups of clients and it seems like the combination of groups and DLM make an interesting match. We leave research into their synergies as future work.

### Declassification

The noninterference property is often too restrictive in practice. Any practically interesting program leaks some kind of acceptable information, e.g. a password checker even leaks information when rejecting a candidate password. To accommodate this intentional leak of information, a way to lower or declassify the security level is needed. Allowing declassification without unintentional release of information has been the focus of recent attention and the paper by Sabelfeld and Sands [82] provides a good survey of declassification. According to the survey downgrading may be classified according to *what* information is revealed (The PER model [80], delimited release [79], relaxed noninterference [61], and quantitative abstractions [27]), by *whom* (The DLM and robust declassification [111]), *where* (non-disclosure [64]), and *when*.

Partial information flow analysis regulates what may be downgraded. The Partial Equivalence Relation (PER) model [80] (a PER on a domain is an equivalence relation on a subset) uses PERs to model the adversary's ability to distinguish between values. The PER model is powerful and captures a wide variety of approaches like delimited release [79], relaxed noninterference [61], and quantitative abstractions [27]. Downgrading in SMCL can quite possibly be formulated in a PER model. The programmer is alerted by the compiler of the possible implicit leaks which may result from a downgrade. An interesting future direction of research is to relate the warnings to the quantitative approach and deduce how much of the information is released.

Who is downgrading the data is important since an adversary may use the declassification mechanism to reveal more secure information than intended. The DLM prevents this by only allowing the owner to downgrade information as specified in the data security labels. This is resembling SMCL where a downgrade can only occur if all server parties (or at least a number of parties equal to the threshold) agree. Another approach is the *robust declassification* by Zdancewic and Myers. [111], where declassification may only be carried out by the designated owner of the data. In a later paper by Myers et al. [67], owner-ship information is used as integrity information, and declassification is deemed safe in areas of high integrity. The connection between integrity and owner-ship is further explored by Zdancewic, who use the DLM extended with integrity labels to determine robust declassifications in [110]. In SMCL all program points are of high integrity since an adversary may alter the computation completely and try to open exactly the secret value he wishes. He will not succeed unless he can corrupt a sufficiently large subset of server parties, in which case he would learn the secret anyway.

The approach based on intransitive noninterference by Mantel and Sands [63] and the non-disclosure approach by Matos and Boudol [64] are similar to downgrading in SMCL. The usual noninterference property does not hold in the presence of declassification, but as observed by both Mantel and Sands and Matos and Boudol the property can be enforced in maximal paths along which there is no declassification, and then we can restart the bisimulation game in the context of any new low-equivalent stores. Our notion of trace security achieves the same goal using similar techniques: localization of declassification and enforcing the identity property between declassifications. The trivial information flow relation is left implicit in SMCL since we only use a two-level security lattice.

Information may be downgraded over time. The SMCR is based on computational security, so the values transmitted from clients to the server are encrypted using public-key cryptography. Thus an adversary may reveal these values if he has sufficient patience to break these cryptographic systems.

Recently there has been some effort to combine the four dimensions of declassification. Askarov and Sabelfeld [11] propose a *localized delimited release* as a combination of the *what* and *where* dimension.

#### Timing Attacks

Timing channels can present a serious threat. The problem of preventing timing attacks has received significant attention, and we will only consider those approaches closely related to SMCL.

Volpano and Smith [102] propose a notion of protected branches with atomic execution time. Their approach guarantees absence of timing leaks observable in the program, but does not prevent external timing leaks and forbid the use of loops in secret conditionals. Agat [5] observed that branches of secret conditionals must have the same timing characteristics in order to prevent timing attacks. Agat proposed to use transformation as a tool to remove timing attacks. In secret conditionals time parameters from the semantics are used to guide a cross padding of instructions with dummy instructions to ensure the same execution time of the two branches. The technique has inspired others like Barbosa and Page [13] who analyze functions (branches) to find the least set of dummy assignments that make their execution time equivalent. In some sense we employ the simplest possible variant of this approach: execute both branches in sequence and join the effects on the store. Our approach is potentially a lot slower than the approach by Barbosa and Page, but we cannot apply dummy assignments because the adversary may inspect the instruction pointer and thus learn which branch is being executed, so to eliminate this possibility we must execute both of the branches.

Tolstrup and Nielson [97] consider VHDL programs for which they define a semantic definition of security against timing attacks based on bisimulation and use a type system to enforce the condition. In SMCL there is no need for transformations and a type system is only needed to prevent loops on secret values. The lack of timing channels is vacuously true due to the semantics of SMCL. The model of Köpf and Basin [59] is a general and abstract semantic model based on automata for observable input and output, which is suitable for many situations but not entirely for SMCL, since the capability of the adversary is not just a function of the input and output, but also of the instruction pointer and state of the computation at any time.

### 1.9.3 Information-Flow Aware Languages

A number of general-purpose programming languages have been extended with support for information-flow security. The decentralized label model has been used as basis for the Java extension *JIF* [66] where the Java type system is extended with labels and principals, however JIF programs are executed under the assumption of a *trusted execution platform* that enforces the rules of the language. This assumption is much to strong for SMC. Other examples are FlowCaml [75] which is an extension of Caml with security-types, and information-flow inference for ML [75]. SMCL is similar to these in the sense of employing security types, but the goal of SMCL is to make it easy to write secure SMC programs.

In [112] Zdancewic proposes secure program partitioning as a mean of allowing mutual distrusting hosts to execute a program. A program is partitioned into a number of slices according to security types and trust declarations. Confidentiality of information is obtained by restricting the computation on values to the host who owns the values or any host trusted by the owner. This has some resemblance to SMCL since both operate in a scenario of untrusted hosts, but whereas program partitioning is aiming at removing the need for a universally trusted host, SMCL realizes such a host through a combination of secure multiparty computation and language-based security. A limitation of program partitioning seems to be that functions on secret values not owned by any of the parties are not possible to compute, without revealing the values to at least one of the parties.

The InCert project [31] suggests to develop a programming language that enables the development of secure applications operating on multiple data sources controlled by different principals without violating the security policies of the involved principals. SMCL may be viewed as realizing parts of this ambitious goal for the special case of the strictest possible security policy where no principal must learn anything about any other.

### 1.9.4 Validation of Cryptographic Protocols

Much effort has been done in the area of validating cryptographic protocols [3,42], and a domain-specific language for verifying such protocols has been proposed by Gordon and Jeffery [45].

A language which tackles similar security threats as SMCL is CAO [12]. CAO is a DSL for cryptographic software and it would be possible to implement the SMCR in CAO, thus ensuring the absence of timing attack at the basic level.

### 1.9.5 Conclusion

We have presented an overview and discussed the relation between SMCL and related work. SMCL is an imperative programming language for general Secure Multiparty Computation in contrast to Fairplay and FairplayMP which are focused on Secure Function Evaluation, and SMC which lacks branches on secret values. SMCL applies techniques from within the area of language-based information-flow security in a novel way. Furthermore we have found no apparent relationship between SMCL and languages for verification of cryptographic protocols, besides that they may be used to verify the implementation of SMCR.

## 1.10 Conclusion and Future Work

We have given a conceptual analysis of the domain of SMC programming in Section 1.2 and identified the following key concepts within the area of SMC programming:

- *Architecture*: The client-server view forms the fundamental computing paradigm of SMC, providing a separation between private, public, and secret computations and between logical and physical parties.
- *Values*: Values are either secret, private, or public, which also determines their runtime representation and separates the efficiency of primitive operations by several orders of magnitude.
- *Communication*: Clients communicate with the server only, either by using tunnels or by reacting to remote procedure calls from the server.
- *Expressiveness*: A general SMC framework must be able to perform any computation; i.e., it must be Turing-complete on private and public values.
- *Security*: Writing reliable SMC programs that do not leak unintended information, is a tedious and error-prone task that can benefit from automated assistance.

In Section 1.3 we described the design and implementation of the Secure Multiparty Computation Language, a high-level, domain-specific language, which allows programmers to express concepts such as clients, server, and operations on secret values directly. We have discussed the basic concepts of SMCL, how they are connected, their restrictions and use. We have presented the seminal Millionaire's problem in SMCL and how it may be generalized in different ways.

We have developed a concurrent semantics, described in Section 1.4, for the entire SMCL language which is carefully crafted to ensure security of SMCL programs, along with the effect-based type system for SMCL which prevents SMCL programs from getting "stuck" and approximate our concept of hoistability. An SMCL program can get "stuck" as the result of an insecure operation. The type system also provides better precision than previous type systems for noninterference, and we prove of its soundness.

The notion of security for SMCL programs is defined in Section 1.6 in terms of the concept of *trace security* which is based on a formalization of physical measurements and we show that SMCL programs are trace-secure under the reasonable assumption that all the operations of runtime we use are trace-secure. We also discuss a simple language of checked annotations for describing potential information dependencies among variables in SMCL programs.

We present an optimization which potentially reduces the number of multiplications on secret values in an SMCL program and prove that it is correct. A multiplication on secret values may be up to 1000 times as slow as multiplications on non-secret values.

As future work we intend to develop a version of SMCL embedded in the Python programming language. Python has been chosen as host language because of its dynamic nature and because we want to take advantage of the VIFF framework [34] which is a new API for SMC written in Python. Furthermore we will investigate ideas for enhanced semantic security which involves automatic generation of a security proof for a given program.

# Chapter 2

## Secure Multiparty Computation (Language) Goes Live

*The best way to predict the future is to invent it*
— Xerox PARC

### 2.1 Introduction

In this Chapter, we present the implementation of a secure system for trading quantities of a certain commodity among many buyers and sellers, a so-called double auction. In the particular case where our system has been deployed, it was used by Danish farmers to trade contracts for sugar beet production on a nation-wide market. The system was implemented using *secure multiparty computation*. This allowed us to ensure that each bid submitted to the auction was kept encrypted from the time it left the bidder's computer, no single party had access to the bids at any time. Nevertheless the system could efficiently compute the price at which contracts should be traded. This was, to the best of our knowledge, the first large-scale practical application of secure multiparty computation.

Below, we first explain the application scenario and the reasons why multiparty computation turned out to be a good solution. We then explain how the SMCL language contrubuted to the solution. Finally, we describe the system that was implemented and report on how it performed.

### 2.2 The Application Scenario

In this section we describe the practical case in which our system has been deployed. In the paper by Bogetof et al. [19], preliminary plans for this scenario and results from a small-scale demo were described.

In Denmark, several thousand farmers produce sugar beets, which are sold to the company Danisco, the only sugar beets processor on the Danish market. Farmers have contracts that give them rights and obligation to deliver a certain amount of beets to Danisco, who pay them according to a pricing scheme that is an integrated part of the contracts. These contracts can be traded between farmers, but trading has historically been very limited and has primarily been done via bilateral negotiations.

In recent years, however, the EU drastically reduced the support for sugar beet production. This and other factors meant that there was now an urgent need to reallocate contracts to farmers where productions pays off best. It was realized that this was best done via a nation-wide exchange, a double auction.

**Market Clearing Price** Details of the particular business case can be found in [15]. Here, we briefly summarize the main points while more details on the actual computation to be done are given later. A double auction includes several buyers and sellers and the goal is to find the so called *market clearing price*, which is a price per unit of the commodity that is traded. What happens is that there is a grid of prices and each buyer places a bid by specifying, for each potential price, how much he is willing to buy at that price. Similarly sellers say how much they are willing to sell at each price.[1] All bids go to an auctioneer, who computes, for each price, the total supply and demand in the market. Since we can assume that supply grows and demand decreases with increasing price, there is a price where total supply equals total demand, and this is the price we are looking for. Finally, all bidders who specified a non-zero amount to trade at the market clearing price get to sell/buy the amount at this price.

**Ensuring Privacy of Bids** A satisfactory implementation of such an auction has to take some security concerns into account: Bids clearly reveal information, e.g. on a farmer's economic position and his productivity, and therefore farmers would be reluctant to accept Danisco acting as auctioneer, given its position in the market. Even if Danisco would never misuse its knowledge of the bids in the ongoing renegotiations of the contracts (including the pricing scheme), the mere fear of this happening could affect the way farmers bid and lead to a suboptimal result of the auction. On the other hand, the entitled quantities in a given contract are administrated by Danisco (and adjusted frequently according to the EU administration) and in some cases the contracts act as security for debt that farmers have to Danisco. Hence running the auction independently of Danisco is not acceptable either. Finally, the solution of delegating the legal and practical responsibility by paying e.g. a consultancy house to be the trusted auctioneer would have been a very expensive solution.

The solution decided on was to implement an electronic double auction, where the role of the auctioneer would be played by three parties, namely representatives for Danisco, DKS (the sugar beet growers' association) and the SIMAP research project (the project in terms of which the work of designing and implementing the solution has been carried out). By interacting with each other, these three parties together could form a "virtual auctioneer", computing the market clearing price and quantities to trade, just as described above. This was implemented using secure multiparty computation technology: each bidder sends his bid in appropriately encrypted form to the three parties, who then compute on the data *while it is still in protected form*. Therefore, no single party ever has access to any bid in the clear. Still, by collaborating, the parties can produce the required output.

A three party solution was selected, partly because it was natural in the given scenario, but also because it allowed using very efficient cryptographic protocols to do the secure computation.

**Motivation** It is interesting to ask what motivated DKS and Danisco to try using such a new and untested technology? One important factor was simply the obvious need for a nation-wide exchange for production rights, which had not existed before, so the opportunity to have a cheap electronic solution – secure or not – was certainly a major reason. We do believe, however, that security also played a role. An on-line survey carried out in connection with the auction showed that farmers do care about keeping their bids private (see tables in 2.1 and 2.2). And if Danisco and DKS would have tried to run the auction using conventional methods, one or more persons would have had to have access to the bids, or control over the system holding the bids in

[1] In real life, a bidder would only specify where the quantity he wants to trade changes, and by how much. The quantities to trade at other prices then follow from this.

cleartext. As a result, some security policy would have had to be agreed, answering questions such as: who should have access to the data and when? who has responsibility if data leaks, and what are the consequences?

Since the parties have conflicting interests, this could have lead to very lengthy discussions, possibly bringing the whole project to a halt. In an interview with the involved decision makers from Danisco and DKS these confidentiality issues were well recognized. Using a consultancy house as mediator would have been more expensive as mentioned, and the parties would still have had to agree on whether the mediator's security policy was satisfactory. As it happened, there was no need for this kind of negotiation, since the multiparty computation ensured that no one needed to have access to bids at any point.[2]

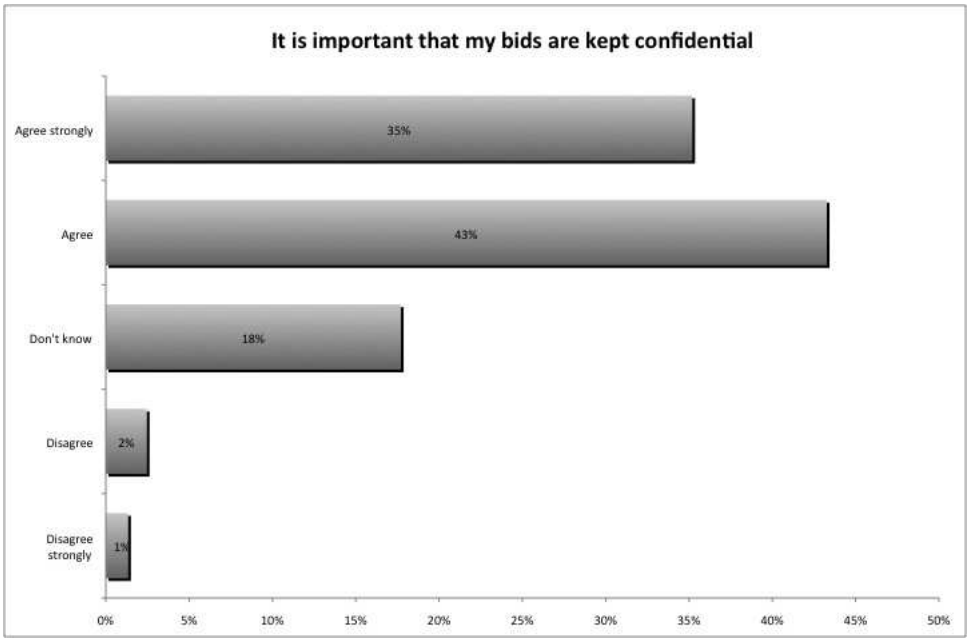



Figure 2.1: Farmers' confidentiality expectations. (Numbers from survey based on questions asked to the farmers after they had submitted their bids.)

**Security and Risks** One must of course consider which attacks such a system might be subjected to, and in particular whether the participants themselves might attack the system. With respect to the three parties doing the secure computation, the situation was as follows: neither party seriously suspected that any of the others would actively and maliciously attack the system. On the other hand, giving all the sensitive data in the clear to one party was not acceptable, and moreover, none of the parties wanted the responsibility of having to store the sensitive data – this would immediately lead to all the practical problems described above, with security policies and procedures.

A suitable solution was therefore a protocol where one assumes that all parties act as they are supposed to, but where no party ever gets access to any sensitive information. This is known as semi-honest security and this is the model we chose for our system. In a nutshell, semi-honest security can be described as a model where one

[2]In an interview with the decision makers, they recognized that this fact made it easy to communicate the security policy to the farmers.

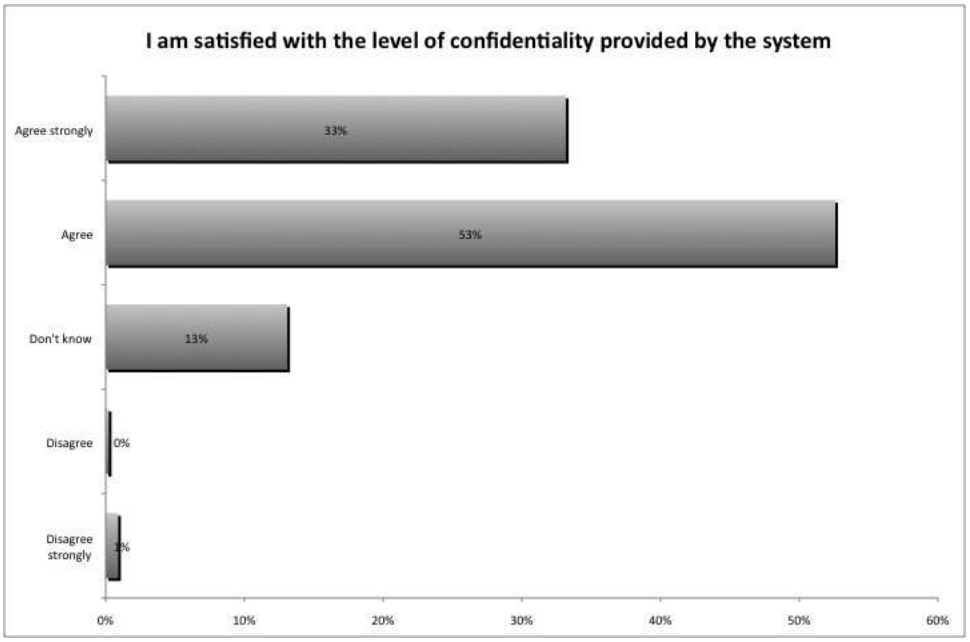



Figure 2.2: Farmers' satisfaction with the provided level of confidentiality. (Numbers from survey based on questions asked to the farmers after they had submitted their bids.)

can "choose not to know" any sensitive data and therefore does not have to assume sole responsibility for keeping them secret.

With respect to malicious attacks from bidders, we estimated that the risk of this happening in our particular case was not large enough to motivate the extra cost of protecting against it: Bidders have a clear interest in the auction working properly, and would anyway have to reverse engineer an applet supplied by the system to even start an attack. Still, in other scenarios, or perhaps in future auctions, malicious attacks from the client side might well be a valid concern, and we therefore show below protocols that protect against malicious bidders.

**Alternative Cryptographic Solutions** One might also ask if the full power of secure multiparty computation was actually needed? Our solution ensures that no single player has any sensitive information, and it might seem that one could solve the problem more efficiently in a similar trust model using a trick often used in voting protocols: one party $P_1$ receives the bids in encrypted form from the bidders, however, the bids are encrypted with the public key of another party $P_2$. Then $P_1$ sends the encryptions, randomized and in permuted order to $P_2$ who decrypts the bids and computes the market clearing price. While this achieves some security because P2 does not know who placed which bids, we have to remember that bids contain much more information than what is conveyed by the result (the market clearing price), e.g. one can see the quantities people were willing to buy or sell at other prices than the clearing price. In principle, this type of information is highly valuable for a monopolist such as Danisco in order to exercise its market power, e.g. in terms of setting the price of an extension or a reduction of the total processing capacity. To what extend such a situation is relevant in practice is not easy to answer. Our conclusion was that using full-blown multiparty computation is a better solution because it frees us from even having to consider the question.

## 2.3 The Auction Implementation

In the system that was deployed, a web server was set up for receiving bids, and three servers were set up for doing the secure computation. Before the auction started, public/private key pairs were generated for the computation servers, and a representative for each involved organization stored the private key material on a USB stick, protected under a password.

Each bidder logged into the webserver and an applet was downloaded to his PC together with the public keys of the computation servers. After the user typed in his bids, the applet secret shared the bids, and encrypted the shares under the computation server's public keys. Finally the entire set of ciphertexts were stored in a database by the webserver.

As for security precautions on the client side, we did not explicitly implement any security against cheating bidders, as mentioned and motivated in a previous section. Moreover, we considered security against third-party attacks on client machines as being the user's responsibility, and so did not explicitly handle this issue.

After the deadline for the auction had passed, the servers were connected to the database and each other, and the market clearing price was securely computed, as well as the quantity each bidder would buy/sell at that price. The representative for each of the involved parties triggered the computation by inserting his USB stick and entering his password on his own machine to access his private key.

The system worked with a set of 4000 possible values for the price, we denote theses the *price indices*, and each price index corresponds to a price. We will freely use the word *price* to refer to the price index associated with the price. The possible values for the price are assumed to be ordered in the natural order from smallest to largest. The upper bound on the number of possible values for the price means that the market clearing price can be found using about 12 secure comparisons.

The bidding phase ran smoothly, with very few technical questions asked by users. The only issue was that the applet on some PC's took up to a minute to complete the encryption of the bids. It is not surprising that the applet needed a non-trivial amount of time, since each bid consisted of 4000 numbers that had to be handled individually. A total of 1229 bidders participated in the auction, each of these had the option of submitting a bid for selling, for buying, or both.

The secure computation we implemented is slightly more complicated than the one we described in a previous section. This is because we have to take into account the possibility that there may not exist a price for which supply matches demand exactly. However, there must exists a maximal price for which demand is at least supply, and likewise a minimal price for which supply is at least demand. These will be an upper, respectively a lower bound on the market clearing price (MCP). The parties doing the computation are told these upper and lower bounds and must then decide what MCP should be, based on these bounds and rules that are agreed on in advance. The computation therefore involves the following steps:

**decrypt to shares (buyers)** the servers shares of buy bids are decrypted

**decrypt to shares (sellers)** the servers shares of sell bids are decrypted

**first search** an upper bound on the MCP is located

**second search** a lower bound on the MCP is located

**marginal search** bids which may help resolve the MCP if the upper and lower bound do not match are located

**open marginal bids** bids which may help resolve the MCP if the upper and lower bound do not match are opened

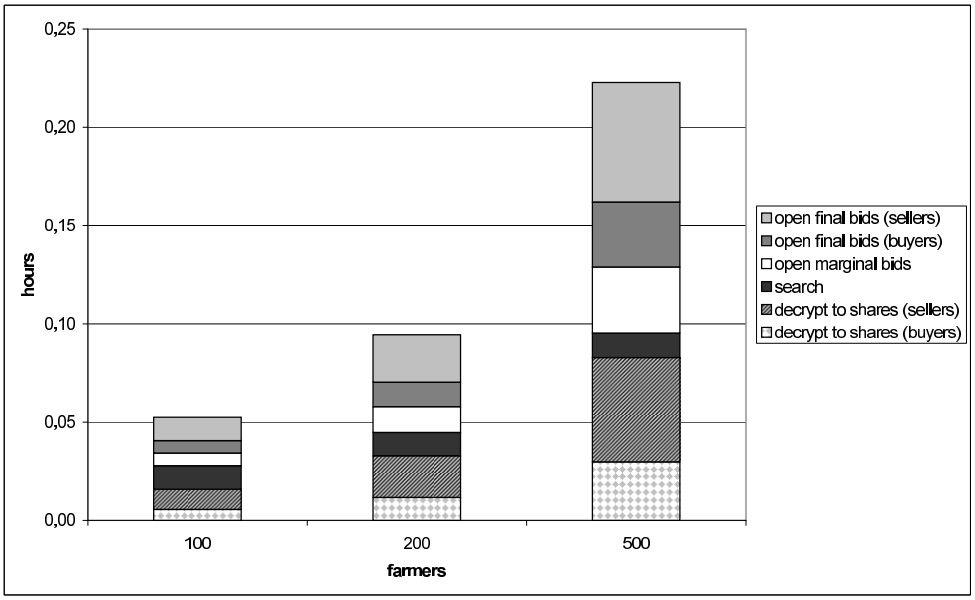



Figure 2.3: Timings

**open final bids (buyers)** for each bidder the buying bids to be realised based on the MCP are opened

**open final bids (sellers)** for each bidder the selling bids to be realised based on the MCP are opened

Figure 2.3 shows how much time is spent on the computation for different sizes of input (we did not time the real auction, as we did not find it appropriate to compute anything else than what was told to the participants. Timings where however performed in exactly the same setup, except that random bids where used). Both the timing runs as well as the actual auction used three Dell laptops (Latitude D630) with 4 GiB RAM (Java being allocated 1500 MiB on each machine), Intel Centrino Dual Core 2.2 GHz processor, running Windows XP Pro, and connected through an Ethernet LAN using a 100 Mbps switch.

In the figure, the number of prices is constant, but the number of bidders vary. Since the number of secure comparisons we need for the search steps only depend on the number of prices, we expect that the time needed to decrypt shares will dominate the time for searching when the number of bidders is large enough. Figure 2.4 shows that this happens when the number of bidders reaches 500. This is not surprising, as the input to the computation, with e.g. 1229 bidders, consist of about 9 million individual numbers.

The lesson to learn here is that the optimized comparison protocol we have developed is efficient enough that it is not a limitation in our scenario, except perhaps in cases with a very small number of bidders. The potential for further optimization therefore lies in the procedure with which shares of bids are encrypted and decrypted. The low-level tools used for this (to do public-key cryptography and pseudo-random functions [44]) were standard off-the-shelf, and there may therefore be faster solutions even without changing the protocols. It should be noted, however, that all this only holds for a double auction (where the number of comparisons does not depend on the number of bidders). For a standard first- or second-price auction, the number of comparisons grows with the number of bidders, and here the comparison time is very critical for a large auction.

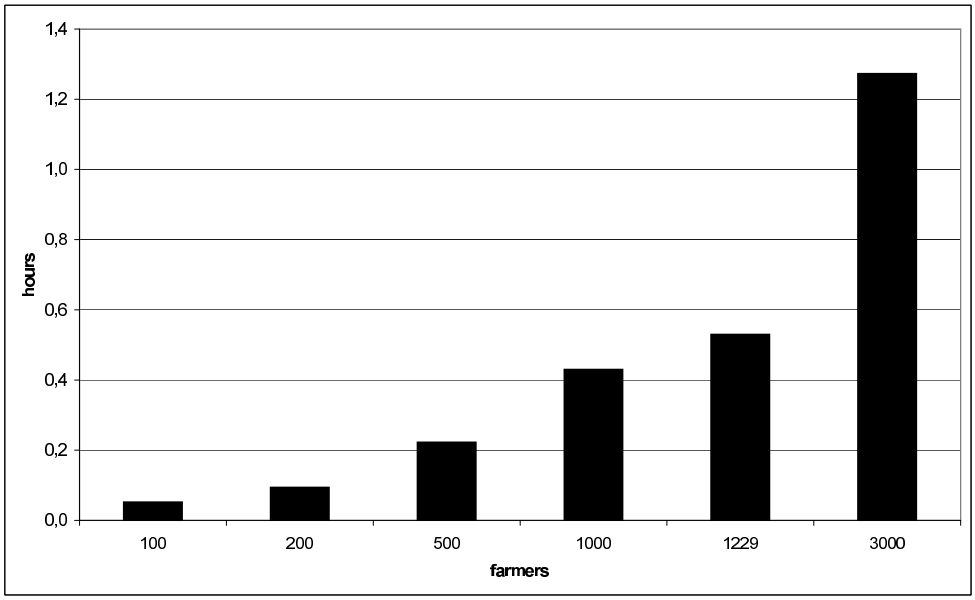



Figure 2.4: Detailed Timings

The actual computation was done January 14, 2008. As a result of the auction, about 25 thousand tons of production rights changed owner. To the best of our knowledge, this was the first large-scale and genuinely practical application of secure multi-party computation.

## 2.4 The SMCL Contribution

In this section we will give an overview of how the double auction logic has been implemented using SMCL. SMCL has been a natural choice as implementation language since it is domain-specific and makes the implementation of the double auction clear and concise, compared to a similar implementation in e.g. Java. SMCL also provides strong security guarantees - any well-typed SMCL program does not reveal information about secret values through its control-flow which prevents side-channel attacks.

The complexity of the application scenario made it infeasible to implement the entire system in SMCL. However it has never been the goal that entire systems should be implemented using SMCL because it is a domain-specific language. The anticipated usage scenario for SMCL has been for SMCL programs to be a part of a larger system. In the case of the double auction we have chosen to implement the core auction logic in SMCL, and implement the rest of the system using a mixture of mainly Java and PHP. The decision is based on the principle of using the best tool to the task, and SMCL is in our opinion by far the best tool for the auction logic since it provides clear syntax and strong security guarantees. However SMCL is not ideal for doing e.g. GUI programming or database operations, in such cases a general purpose language like Java is better suited.

We start by describing the actual auction logic in more detail, which will help understand the implementation. Figure 2.5 shows an example of aggregated demand and aggregated supply as a function of the price indices, remember that the indices range from 0 to 3999. The aggregated demand $agrDemand[i]$ or aggregated supply $agrSupply[i]$ for a price index $i$ is the sum of the demand or supply at that price from each farmer. Note that the aggregated demand is monotone decreasing function and the aggregated supply is a monotone increasing function.

The goal of the auction is to identify the market clearing price, which is where demand meets supply, i.e. the price where the aggregated demand equals (or is closest to) the aggregated supply. The MCP is located in the interval denoted $X'$ in Figure 2.5.

To identify the interval $X'$ we identify the two boundaries *p1* and *p2*. We identify them by first defining the excess supply $z[i]$ as the difference in the aggregated demand and supply at price index $i$: $z[i] = agrDemand[i] - agrSupply[i]$. The excess supply is a monotone decreasing function of the price, because the demand diminishes and the supply increases as the price goes up. Which means that the excess supply will have the maximum value for the smallest price indices and the minimum value for the highest indices.

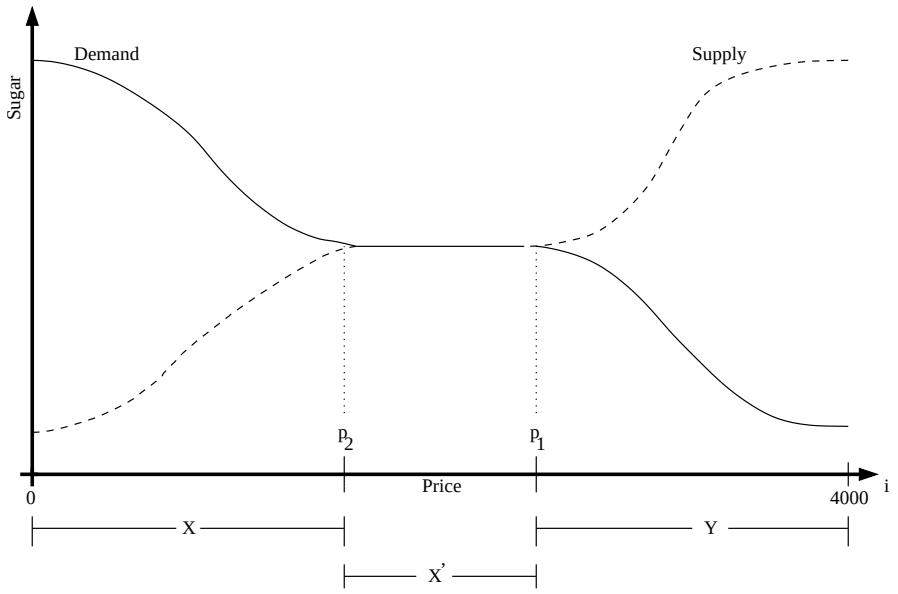



Figure 2.5: Aggregated demand and supply displayed as a function the price indices $i$

Now *p1* is the upper bound on the market clearing price. *z1* is the excess supply at the price index *p1*. *p2* is a lower bound on the market clearing price. *z2* is the excess supply at the price index *p2*.

Definition of *p1* and *z1*. Let $X$ be the subset of price indices $\{1, \ldots, 4000\}$ where $z[i]$ is zero or positive. Let $i_{min}$ be the element in $X$ such that $z[i_{min}]$ is minimal in $\{z[i] \mid i \in X\}$ and define $X' = \{i \in X \mid z[i] = z[i_{min}]\}$. Then we define $p1 = max(X')$, thus *p1* is an upper bound on the price index in $X'$. We also define $z1 = z[p1]$. $X$ might be empty in the case where the supply exceeds the demand, in which case *p1* and *z1* are not defined.

Definition of *p2* and *x2*. Let $Y$ be the subset of the indices $\{1, \ldots, 4000\}$ where $z[i]$ is less than or equal to zero. Let $i_{max}$ be the element in $Y$ such that $z[i_{max}]$ is maximal in $\{z[i] \mid i \in Y\}$ and define $Y' = \{i \in Y \mid z[i] = z[i_{max}]\}$. Then we define $p2 = min(Y')$, which means that *p2* is an lower bound on the price index in $Y'$. We also define $z2 = z[p2]$. $Y$ might be empty in the case where the demand exceeds the supply, in which case *p2* and *z2* are not defined.

The market clearing price might not be unique but can be an entire interval as indicated in Figure 2.5, where all the prices in the interval $X'$ are candidates to be the MCP. We denote these candidates as the *marginal prices*. The marginal prices

will be the same no matter which we chose. However, the consequences might be very different, because there might be large differences in how the aggregated demand and supply are structured in terms of contracts. E.g. is it a few big contracts sold at one price index and many small contracts at another, or how large amount of productions rights are traded at given prices. These are political issues that needed to be resolved prior to running the auction. The policy agreed upon, where used in an interactive iterative process where the bids for *some* of the marginal prices where revealed to all parties, and a final market clearing price index was chosen. The process of selecting which marginal prices will be the market clearing price will not be described in details.

We have found it useful to add an ad-hoc language extension to SMCL to make the integration with the Java based SMCR runtime simpler. We have added *single-inheritance class based objects* to SMCL, as exemplified in Figure 2.6 where we define a number of classes that will be used as datastructures in the double auction. A *class* in extended SMCL corresponds directly to a class in Java, except that we can use the usual SMCL datatypes in classes as well, e.g. in Contract where the *field amount* has the SMCL type sint. The addition of classes does not affect the security properties of SMCL because classes can be straight forwardly encoded in SMCL using records.

The classes in Figure 2.6 defines the concept Farmer which is the super type of a SellingFarmer and a BuyingFarmer, each representing a farmer who wants to sell and buy sugar beet contracts respectively. A Farmer has a name which identifies him uniquely. A SellingFarmer has an array of SellBids called *bids*, there is a bid for each of the 4000 possible values of the price, and a SellBid consists of the *amount* and the id of the contract to be sold at that price if it is chosen as the marked clearing price. Both the amount and the contract id is represented as secret values because the farmer does not want to reveal which contracts are sold at which price. A BuyingFarmer has an array of sints called *bids*, one bid for each of the 4000 possible values of the price. A bid describes how many tons of productions rights the farmer would like to buy at the specified price.

```
class abstract Farmer {
  String name;
}
class SellingFarmer
    extends Farmer {
  SellBid[] bids;
}
class SellBid {
  sint amount;
  sint contractId;
}
class BuyingFarmer
    extends Farmer {
  sint[] bids;
}
```

```
class AuctionResult {
  int mcp;
  int p1;
  int p2;
  int p3;
  int z1;
  int z2;
  int z3;
  int k;
  Contract[] contracts;
}
class Contract {
  int id;
  int amount;
}
```

Figure 2.6: Classes used in the Double Auction example

The double auction is implemented as the `server` DoubleAuction shown in Figure 2.7. Here we define the tunnels *buyingFarmers* and *sellingFarmers*, which will be connected to a database containing the bids from farmers as described in Section 2.3.

The server also contains the *main* function where execution of the server begins and a number of auxiliary functions used to implement various parts of the auction. In this section we will only focus on the main auction logic and we will elide parts of functions where possible for brevity and clarity. The entire implementation is provided in the Appendix A without further comments.

```
declare server DoubleAuction: {
  tunnel of BuyingFarmer buyingFarmers;
  tunnel of SellingFarmer sellingFarmers;
  ...
```

Figure 2.7: Double Auction implemented in SMCL

A part of the main function of the server is shown in Figure 2.8. We first compute the aggregated demand and supply, then search for the market clearing price, and finish with a search for the marginal bids. The computation of the aggregated demand and supply is straight forward and can be found in Appendix A.3.

```
function void main: {
  sint[ ] agrDemand = computeAgrDemand();
  sint[ ] agrSupply = computeAgrSupply();
  AuctionResult res = searchPrice(agrDemand, agrSupply);
  // compute marginal bids
  ...
  return;
}
```

Figure 2.8: Main function of the Double Auction server

The upper and lower bounds *P1*, *P2* are computed by the function *searchPrice* shown partly in Figure 2.9. We have elided the part of function which computes various values used for bookkeeping and auditing for clarity. *searchPrice* uses the functions *search1* and *search2* to compute *s1* and *s2* which are candidates for *p1* and *p2* respectively. Both *search1* and *search2* work in similar ways so we only show *search1* in Figure 2.10 and provide *search2* in the appendix as Figure A.8. *search1* does binary search and returns the maximal price index such that the excess supply is zero or positive. If the excess supply is less than zero for all indices then zero is returned. Notice that we compare the value of the aggregated demand and the aggregated supply secretly, so we don't reveal information about the demand or supply. We open the result of the comparison to make the search faster. This is allowed since any of the parties involved will be able to deduce this information from the MCP, since the difference between demand and supply is monotone.

When we have computed *s1* and *s2* we have to handle a number of special cases, depending on for how many indices the excess supply is zero. The cases can be categorized into two overall categories: $s1 = s2$ and $s1 != s2$. In the first case *s1* is equal to *s2* if there is exactly one index where the excess supply is zero, or if there are no such indices and the excess supply is either positive or negative for all indicies. Which is handled by line 6-19 in Figure 2.9. If the excess supply is zero then save *p1*, *z1*, *p2*, *z2* accordingly. If the excess supply is positive then the excess supply is positive for all indices, and *p2* and *z2* are undefined. Similar if the excess supply is negative,

```
function AuctionResult searchPrice(sint[ ] agrDemand,
                                   sint[ ] agrSupply) {
  AuctionResult res = new AuctionResult();
  int s1 = search1(agrDemand, agrSupply);
  int s2 = search2(agrDemand, agrSupply);
  if (s1 == s2) {
6:  int tz = open(agrDemand[s1] − agrSupply[s1] |
                  agrDemand, agrSupply, s1);
    if (tz == 0) {
      res.setP1(s1);
      res.setP2(s1);
      res.setZ1(tz);
      res.setZ2(tz);
    } else if (tz > 0) {
      res.setP1(s1); // p2, z2 undefined
      res.setZ1(tz);
    } else {
      res.setP2(s1); // p1, z1 undefined
      res.setZ2(tz);
19:  }
  } else {
    res.setP1(s1);
    res.setP2(s2);
    res.setZ1(open(agrDemand[s1] − agrSupply[s1] |
                   agrDemand, agrSupply, s1));
    res.setZ2(open(agrDemand[s2] − agrSupply[s1] |
                   agrDemand, agrSupply, s2));
  }
  ...
  return res;
}
```

Figure 2.9: The searchPrice function computes the indicies of the potential

then *p1* and *z1* are undefined.

In the second case where $s1 != s2$ there might no or more than one indices at which the excess supply is zero. In either case we save *p1* and *p2*, and open the excess supply, and save it as well.

Figure 2.11 shows the signatures for some of the functions mentioned in the code. We have omitted their definitions, since their implementation relies on a tight integration with Java for doing user input.

```
function int search1(sint[] agrDemand, sint[] agrSupply) {
  int low = 1;
  int high = agrSupply.length + 1;
  while (low + 1 < high) {
    int mid = (low + high + 1) / 2;
    sbool res = agrDemand[mid - 1] == (agrSupply[mid - 1]);
    if (open(res | res)) {
      low = mid;
    } else {
      high = mid;
    }
  }
  return low - 1;
}
```

Figure 2.10: Binary search for the upper bound on the index of the market clearing price

```
function void promptComputeMarginal(AuctionResult res) {}
function void openMarginalBids(int k, int base,
                               AuctionResult res, sint[] agrDemand) {}
```

Figure 2.11: Function signatures for tightly integrated functions

## 2.5 Conclusion

How successful have we been with the auction system and does the technology have further potential in practice?

Other than the fact that the system worked and produced correct results, it is noteworthy that about 80% of the respondents in an on-line survey said that it was important to them that the bids were kept confidential 2.1, and also that they were happy about the confidentiality that the system offered. Of course, one should not interpret this as support for the particular technical solution we chose, most farmers would not have any idea what multiparty computation is. But it is nevertheless interesting that confidentiality is seen as important. While it is sometimes claimed that ordinary people do not care about security, we believe our experience shows that they sometimes do care. Our impression is that this has to do with the fact that money is involved, and also that other parties are involved with interests that clearly conflict with yours. For instance, given the history of the sugar beet market, there is little doubt that "confidentiality" for the farmers include confidentiality against Danisco. Danisco and DKS have been satisfied with the system, and at the time of writing, the auction has already been run successfully a second time.

During the experiment we have therefore become convinced that the ability of multiparty computation to keep secret *everything* that is not intended to be public, really is useful in practice. As discussed earlier, it short-circuits discussions and concerns about which parts of the data are sensitive and what common security policy one should have for handling such data.

It is sometimes claimed that the same effect can be achieved by using secure hard-

ware: just send all input data privately to the device which then does the computation internally, and outputs the result. Superficially, this may seem to be a very simple solution that also keeps all private data private. Taking a closer look, however, it is not difficult to see that the hardware solution achieves something fundamentally different from what multiparty computation does, *even if one believes that the physical protection cannot be broken*: Note that we are still in a situation where some component of our system – the hardware box – has access to *all* private data in cleartext. If we had been talking about an abstract ideal functionality, this would – by definition – not be a problem. But a real hardware box is a system component like any other: it must be securely installed, administrated, updated, backed up, etc. In this sense the secure hardware solution is not fundamentally different from a solution using an ordinary central server to receive the bids and do the computation. In particular, both solutions have a component which becomes a single point of attack, and may therefore be less robust than a distributed solution against outsider attacks. Also, both solutions have all the practical problems we pointed out earlier with agreeing on common procedures and security policies if parties have conflicting interests.

We believe that a much more natural use of secure hardware is for each party in a multiparty computation to use it in order to improve *his own* security, i.e. to make sure that the protocol messages is the only data his system leaks.

Another standard alternative to SMC is to pay a trusted party such as a consultancy house to do the computation. We said earlier that the parties in our scenario decided against this because it would have been much more expensive. One could claim, of course, that this was only because the alternative was to have a research team do the whole thing for free – and that hence the experiment does not show that SMC is commercially viable. While the experiment has certainly not produced a business plan, we wish to point out that a SMC based solution only has to be developed once and costs can then be amortized over many applications. In some cases one may not even need to adapt the system – for instance, in the case of the sugar beet auction it is very likely that the same auction will be run once a year for some time to come.

In conclusion, we expect that multiparty computation will turn out to be useful in many practical scenarios in the future.

# Chapter 3

## The Future of SMCL - PySMCL

*Civilization advances by extending the number of important operations we can perform without thinking*
— Alfred North Whitehead

### 3.1 Introduction

In this Chapter we present the current status on our preliminary work on an embedded domain-specific language for secure multiparty computation called PySMCL. PySMCL is embedded in the Python programming language and is designed based on our experience with the domain-specific language SMCL described in Chapter 1.

The goal of PySMCL is to provide and further develop the advantages of SMCL in a fully fledged programming language. Python has been chosen as host language because of its dynamic nature which we exploit in the implementation of PySMCL and because we want to take advantage of the VIFF framework [34] which is a new API for SMC written in Python.

SMCL guarantees that computations performed by SMCL programs does not reveal secret information. However there is still room for improvement. An SMCL program can be seen as a protocol which UC-realizes some ideal functionality in the sense of the Universally Composable (UC) model [24], which is a cryptographic proof framework. In the UC model one defines what is called an *ideal functionality* which is a specification of the computation to be done along with a definition of exactly which information should be reveal and to who. The ideal functionality reveals by definition exactly the intended information and nothing more. In the UC model, one also define a *concrete protocol*, which is said to UC-realize an ideal functionality if there is a so called *simulation proof* that the protocol reveals exactly the information revealed by the ideal functionality. Our ultimate goal is to automatically generate such proof from a PySMCL program, and the proof should be verifiable by and an automatic theorem prover like HOL [93], Isabelle [94], or Coq [92].

In this Chapter we present the challenges in doing so in the context of PySMCL and we describe our preliminary solutions to these challenges, which includes plans for a static verifier and a preprocessor for the language.

The rest of the Chapter is organized as follows. Section 3.2 describes how we envision PySMCL will be used. Section 3.3 gives a high-level description of the language and we present some example programs. Section 3.4 discuss the security guarantees that the static verifier should enforce. In Section 3.5 we present our implementation strategy. In Section 3.6 we define a core calculus for PySMCL, and we conclude and discuss future work in Section 3.7.

## 3.2 Usage Scenario

The programmer writes his program in PySMCL specifying a desired computation using high-level constructs specialized for the SMC domain, without considering low-level VIFF or SMC protocol details. The programmer then runs the PySMCL preprocessor and verifier which perform static analysis and turns the PySMCL code into Python code which uses VIFF. The programmer will annotate functions which uses SMC constructs with small Python annotations which basically instructs the preprocessor to process the function. In this way we only process functions which uses PySMCL, and as we will show in Section 3.5 we can preserve the dynamic nature one would expect from a DSL embedded in Python.

The verifier will be tracking the flow of secret and random values, and generating warnings for the programmer if information is unintentionally revealed.

If the verifier will provide feedback to the programmer if it finds potential security problems. The feedback could include help towards fixing the problem, and/or help towards proving that the issue in question is actually not a security problem. A help could be proof stub with premises and conclusion to be filled out by the programmer and supplied with the rest of the autogenerated proof to a proof assistant. If the verifier does not find errors, the translated program is executed as an ordinary Python program which uses the VIFF framework.

A PySMCL program is generally executed by a number of parties who will collaborate on carring out the specified computation – a PySMCL program is similar to the server in a SMCL program. We use a configuration file to specify where the parties will be located (e.g. their IP numbers) and which variant of the VIFF run-time to use (that is, which concrete SMC protocols should be used). The file must be supplied to the preprocessor, and may also define additional information as we will discuss in Section 3.4.

## 3.3 The Language

In this Section we informally describe the language by means of two examples. The seminal *Millionaire's problem* [109] and the *Cake bringer* example.

The Millionaire's problem involves a number of millionaires who want to find out which is richer, but all of them refuse to disclose their net worth. We implement a solution to the problem in PySMCL.

```
01: @secure_function
02: def who_is_richest():
03:   millionaires = Group(ConfigFile)
04:   max = 0
05:   rich = Id()
06:   for millionaire in Secret(millionaires):
07:     netWorth = millionaire.get("How much money have you got?")
08:     if (netWorth > max):
09:       max = netWorth
10:       rich = millionaire
11:   return openresult(rich)
```

Figure 3.1: The generalized Millionaire's problem in PySMCL

The implementation is shown in Figure 3.1 where we define the secure function `who_is_richest`. PySMCL will be a syntactic subset of Python which gives us the

advantage that all the tools for manipulating Python code can be reused and the development time of the PySMCL infrastructure can be reduced. We use Python annotations like `@secure_function` to specify *secure functions* which should be preprocessed and verified as PySMCL code. In secure functions we also allow the programmer to create domain-specific values like `Group`s of participants or secret `Secret` values. We implement these *value constructors* as Pythons object instantiation calls or function calls, and the preprocessor will add suitable definitions for the objects and functions. The set of annotations and domain-specific value constructors is not fixed at this stage, but will be developed as needed, as a result of writing test programs for concrete applications during the design and implementation of PySMCL. We will present a model of the language in Section 3.6.

In line `03` we define the variable `millionaires` which is initialized with a group of millionaires. The identifier `ConfigFile` refers to a configuration file which defines a number of identities (IP-addresses, etc.) for the parties who will be in the group. An identity is implemented as a class which provides some methods for communicating with a party. We define the variables `max` and `rich` on line `04` and `05`, and initialized them to zero and the *empty* identity `Id()`, respectively. The empty identity is the identity of nobody.

We now iterate through the group of millionaires, we do this in a secret manner where we keep the identity of the millionaire secret. We can keep the identity of the millionaire secret by having all the millionaires send a value and only keep the right one, as described in Section 1.3. For each `millionaire` we ask him how much money he has, and the millionaire inputs a number which is stored in the variable `netWorth`. We call the method `get` on the millionaire identity object which will make the supplied value provided by the party secret according to the protocol specified in `ConfigFile`. There is also a `public_get` method which does not make the value secret. The verifier will apply a flow sensitive analysis which will track secret values through the program and thus mark `netWorth` as secret because the result of `get` is secret.

For each millionaire we check whether he is currently the richest and if so, we update the `max` and the `rich` variables, which both becomes secret. The verifier will check that the comparison will be a boolean. We need to treat the conditional specially because the check `netWorth > max` is secret, and if handled improperly the conditional might leak information about the `netWorth` and `max`. We denote such conditionals as *secret conditionals*. We use the same technique as in SMCL where we execute both branches, and recombine the results using conditional assignment. Line `09` of the secret conditional will be rewritten to `max=b1*netWorth+(1-b1)*max`, where `b1` is the result of the comparison. Like in SMCL the verifier will be able to check for side-effects in the branches and reject programs with side-effects in branch of secret conditionals.

We use the function `openresult` to disclose the identities in `rich` to all the millionaires who then learns which one is the richest, as intended. We also provide the `open` function which also reveals the value of its argument, but is treated differently by the verifier. `openresult` is treated as revealing the intended result of the function which corresponds exactly to the value computed by the corresponding ideal functionality. But as shown in Section 1.7 we can get significant speed improvements by revealing some information during the computation. Whether the function now UC-realizes the intended ideal functionality now relies on the programmer who should be able to prove that the revealed information can be computed from the result of the ideal functionality. The `open` function is for such cases, where we want to reveal some information which is not the result of the function in the UC sense but can be computed based on the result. The verifier will create proof stubs for each use of an `open` function which should be filled out by the programmer. The stubs can be given to a proof assistant for automatic verification along with the rest of the proof which proves that the function UC-realizes the functionality specified by the function arguments and the `openresult`

calls.

We integrate secure functions with the rest of Python by allowing calls to Python function which does not pass secret values and calls to other secure functions which can be verified. Multiplying random values "hides" secret values, and may be revealed without compromising the secret value. The exact rules for hiding are to be worked out in the final version of PySMCL. They depend on the nature of the underlying runtime, and could also depend on an interval analysis of the values.

PySMCL provides secret variants of the usual operations on integers and booleans, and we use operator overloading in Python to get the usual syntax for these.

PySMCL provides a unified interface to the different variants of the VIFF runtime. These variants implement different SMC protocols, so the unified interface means that the programmer does not need to know any details on how the cryptographic protocols are implemented.

### 3.3.1 The Cake Bringer Example

Imagine a group of cryptographers having weekly meetings. At every meeting one of the participants is supposed to bring a cake.

Some people may feel "more willing" to bring a cake on a certain day, and thus we will choose who brings the cake at random among those persons. But if nobody feels "more willing", then a random person is chosen among everybody. Of course, the willingness of an individual is strictly confidential, and no-one should be able to influence the randomness of the choice.

Note that the purpose of such a computation could also be more serious, such as selecting a volunteer among a set of participants for doing a certain task that must be done, but where we do not want to expose in public people's willingness to volunteer.

Below follows a PySMCL program for this task. The identifier *ConfigFile* is assumed to point to a configuration file that contains the actual identities (e.g. IP-numbers) of the parties in the group called *participants*.

```
@secure_function
def who_brings_cake():
    participants = Group(ConfigFile)
    willing = []
    for participant in participants:
        willing.append(participant.get("willing"))
    how_many_willing = sum(willing)
    if how_many_willing == 0:
        cake_bringer = choose_random(participants)
    else:
        cake_bringer_nr = rand() % how_many_willing
        count = 0
        for (i, w) in enumerate(willing):
            if w:
                count += 1
                if cake_bringer_nr == count:
                    cake_bringer = i
    return openresult(cake_bringer)
```

## 3.4 The Verifier

In this Section we give a high-level description of which security guarantees the verifier is expected to ensure and how it is expected to provide them.

We expect the verifier to give the same security guarantees as SMCL, verified and preprocessed programs does not leak information through the computation. The verifier should ensure that verified programs does not end in a configuration, where no *domain-specific* evaluation rule apply and which is a not a value. By this we mean that domain-specific evaluations should not get "stuck", however we have no intention on ensuring the same for the entire Python language. We believe this gives a good balance between the dynamic nature of Python and our intended security properties for PySMCL programs. The verifier will also ensure that verified programs have no side-effects in the branches of secret conditionals. Side-effects in secret conditionals like I/O or assignment to public variables (implicit flow [36]) are a security threat. Furthermore the verifier should take secret random values into account. Random values can be use for hiding secret values.

The verifier should also be able to provide semantic security, where information is unintentionally leaked through the `open` function. We intend to do so by giving the programmer feedback when information is opened. There are three possibilities:

- The opening is marked as intentional, by using the `openresult` function. This should be done when the opened value is a part of the result that the programmer intends the program to compute. Such openings are always considered valid.
- The opening is done using the `open` function, and the verifier can determine that the opening is harmless, for instance if an information flow analysis shows that due to hiding with random numbers, the opened value contains no sensitive information. In this case the verifier will do nothing further.
- The opening is done using the `open` function, but the verifier cannot determine that the opening is harmless. The verifier interprets this to mean that the programmer claims the opening is harmless. In technical terms, this means that the programmer claims that the opened value can be simulated with an indistinguishable distribution, based only on the intended results of the program. More concretely, this can be the case, for instance, if the opened value directly follows from the intended results (but the programmer gets an efficiency improvement from revealing the value at an earlier stage).

  In such a case, the verifier will generate a warning, informing the programmer about the proof burden that has to be carried out in order for the opening to be proved secure, and providing as much help as possible towards giving such a proof. This help can include an overview of the intended results and a summary of the information flow analysis. Our ultimate goal is to let the verifier generate a (semi) automatic proof that the entire program is secure, where the programmer will fill out the proof stubs generated for these cases of the `open` function.

The verifier does not have explicit type annotation available as in SMCL. Thus the verifier performs a static conservative approximating analysis making a distinction between, and following the data flow through the control flow graph of secret vs. public values, input vs. output, random values, and integers. Within integers we also want a distinction between integers used to represent booleans, and actual integers. In PySMCL the boolean values `true` and `false` are represented as the integers `0` and `1`. To make the distinction we imagine using a variant of range analysis [72].

In PySMCL, one can define groups of parties, and assign a group to a given identifier. This is useful when iterating through all parties in a group and perform "the same" operation for all of them. For an example of this, refer to the "Millionaire's example" in Section 3.1. A group is a set of abstract identities at compile-time, and are concretely defined in a configuration file when the program is run.

The group notion also introduces a possibility to refine the concept of what is public and secret. More precisely, one can reveal a value to *only* a certain group of parties.

As usual, this is considered secure if the programmer specifies that revealing this value to the group is part of the purpose of the program.

On the other hand, the value in question should remain unknown to parties outside the group, and the verifier must take this into account. For instance, it is a potential security breach if variable `a` is opened only to group `A` and later `b` which depends on `a` is opened to everyone. The verifier will therefore include A group analysis which keeps track of which groups of parties know about which values.

In some applications it is useful to work with the full generality of the VIFF framework. A typical example would be a situation where a set of secret values are stored in a database in encrypted form and each party needs to retrieve its shares of the secret values. This is the case, for instance, for some types of auctions [16]. In such an application, we would retrieve the input from the database, convert it to the representation used in the protocol and then proceed as usual. To accommodate for this, PySMCL allows functions to be annotated as *semi-secret functions* which are not verified in the same way as secret functions are, but may use low-level primitives from VIFF and Python in general to construct the result. The verifier will assume that the result is a secret value and the programmer takes the full responsibility for the security of the function. The secret value returned must of cause be internally represented in whatever form the underlying protocol uses for secret data.

## 3.5 Implementation Strategy

In this Section we describe how we intend to implement the preprocessor and the verifier. The implementation will make use of *function decorations* as shown in Figure 3.2. A function decorator like `decorator` is a higher order function which receives the function `f` as arguments and returns a possibly different function which is then named `f`. This is often used for wrapping functions with logging hooks etc.

We will use function decorators in combination with a careful choice of how domain-specific values are constructed to implement the functionality of PySMCL fully in Python, and letting PySMCL programs be syntactically valid Python programs, in order to minimize the learning curve, and to utilize Python's built-in parser. We implement a function decorator `@secure_function` which uses the *inspect* module (available in Python) to get access to the source code and thereby the parse tree of the decorated function. From the parse tree we construct a control flow graph for the various analysis and then translate the tree into Python code which uses the VIFF framework and then returns the translated tree. The translation will translate primitive operations into appropriate function calls in the VIFF framework, perform optimizations on the arithmetic network, and desugar certain expressions, for example turning secret conditionals into a sequence of statements that computes both branches secretly.

We expect to use (at least) the decorators `@secure_function` and `@return_secret` where the first is used for functions that should be subject to verification, and the second is used for functions that return secret values but should not be verified, as described in Section 3.4.

```
@decorator
def f():
    pass
```

Figure 3.2: An example of a function decorator in Python

Using function decorators also has the advantage that using PySMCL functionality

will not require any extra compilation step, as the verification and preprocessing is done dynamically as the file is loaded by the Python interpreter. However, we will provide a special program that one can run on the program files for querying the verification engine about what it has inferred at different program points without actually running the program. This will also give the programmer feedback on which values the verifier has inferred should be secret. This information will usually be necessary in order to remove security problems from a program before running it on data that are actually security-critical.

## 3.6 Core PySMCL

Reasoning about formal properties of PySMCL programs is a daunting and complex task, because PySMCL is embedded in the large and dynamic Python language. Python provides many features which are useful when writing PySMCL programs, like interfacing to external data sources or visualizing output on the screen. Many of these features are not particularly important when it comes to the properties of the new features introduced by PySMCL and how they interact.

We have chosen to alleviate this problem by considering a smaller and more tractable model of PySMCL, which we call Core PySMCL. In this Section we present an initial version of Core PySMCL. The calculus can be a future vehicle for formal reasoning over various properties of PySMCL, like formalization of the different analysis we have proposed in Section 3.4.

The abstract syntax of the calculus is a syntactical subset of Python. We use the notational shorthand $\overline{e}$ which means a possibly comma separated list of expressions: $e_1, \ldots, e_n$. It will be clear from the context if commas should be present or not.

Core PySMCL is based on a model, where we have a set of parties which carries out the computation. The abstract syntax of Core PySMCL is presented in Figure 3.3. A Core PySMCL program consists of a sequence of declarations, and a statement. Declarations are either a class, a group, or an identity declaration. A class declaration is similar to a class declaration in Python. Group and identity declarations describe the identities of the participating parties.

The set of statements and expressions is a standard while-language augmented with domain-specific operations like constructors and destructors of secret values: `Secret`$(e)$, $e_1$.get$(e_2)$, `open`$(e_1,e_2)$, and `openresult`$(e_1,e_2)$.

In Figure 3.3, the intuition/semantics is briefly hinted in the comments. Below, we (informally) describe the semantics in more detail.

### 3.6.1 Semantics

The execution of the program is initiated by evaluating the declarations followed by the statement.

Group and identity declarations refer to a configuration file which maps groups and identities to actual machines and specifies which cryptographic protocol should be used, e.g. Shamir secret sharing.

When executing the statement, we assume that classes `Secret`, `Group`, and `Id` are implicitly defined, and support the operations on them defined by Figure 3.3. We also assume that the functions `open`, `openresult`, `srand`, and `rand` have been defined in a suitable way.

To evaluate `Secret`$(e)$, we first evaluate $e$, which should be a number. The number is then made secret according to the protocol specified for the program. Similar, if $e_1$ in $e_1$.get$(e_2)$ evaluates to an identity, the computation specified by $e_2$ is carried out by that party and the result is made secret according to the protocol for the program.

`open`($e_1$,$e_2$) and `openresult`($e_1$,$e_2$) are both inverses to $e_1$.get($e_2$) in the sense that if $e_1$ evaluates to a value then the value is made public only to the group computed by evaluating $e_2$. If $e_2$ is not specified, then the value is made public to everybody. Although the two open functions are the same syntactically and semantically, they are considered different by the verifier, which consider (only) values opened by `openresult` as part of the intended result from the program which are always legal to open.

Operations on groups are set inclusion ($e_1$.in($e_2$)) and set union ($e_1$.union($e_2$)). How a secret random value is computed by `srand`() depends on the specified protocol.

### 3.6.2 Execution Order

A large part of the PySMCL semantics is inherited from the semantics of Python and SMCL. However, both of them have a strictly sequential execution order of statements which is not the case in PySMCL. PySMCL is translated into Python code which uses the VIFF framework, and VIFF is based on asynchronous communication among the parties. Moreover it will schedule as many computations as possible to be executed concurrently, except when explicitly asked not to (see [34] for details). This means that even if we write

```
a = b+c
x = z*y
```

then it does not necessarily mean that `a` will be assigned a value before `x`. VIFF gets a major efficiency advantage from this behavior, so unless we sacrifice this advantage, we cannot guarantee that all computations are executed in exactly the order they were specified.

Our solution to this problem is to leave the scheduling to VIFF except in cases where it is known to make a difference to security, or perhaps if the programmer asks for an explicit timing.

More precisely, the security properties of a secure program depend only on the order in which it is specified to take input and produce output, and on the way the outputs depend on the inputs. Therefore, a PySMCL program is guaranteed to execute input and open statements in the order they are specified, where, however, *consecutive* openings or consecutive inputs may be done in parallel. We also guarantee, of course, that the outputs are the correct (possibly randomized) function of the inputs.

We will implement possibilities for the programmer to specify a specific required ordering of other types of statements, if future work on the implementation suggests that this is a feature we need.

### 3.6.3 Example

Figure 3.4 presents an example program written in Core PySMCL. In the example we first define the identities of party $A$ and $B$ and then we retrieve secret values from both using the *get* method and store them in variables $a$ and $b$ respectively. Here ... abbreviates code which allows the party to input a value. We then compute whether $a$ is bigger than $b$, and open the result to the group of $A$ and $B$.

| | | | |
|---|---|---|---|
| $x$ | | | identifier |
| $C$ | | | class name |
| $P$ | ::= | $\overline{d}\ S$ | program |
| $d$ | ::= | `class` $C\,(\overline{x})$: $\overline{md}$ | class definition |
| | \| | $x_1$ = `Group`$\,(x_2)$ | group declaration |
| | \| | $x_1$ = `Id`$\,(x_2)$ | identity declaration |
| $md$ | ::= | `def` $f\,(\overline{x})$: $S$ | method definition |
| $S$ | ::= | $x = e$ | assignment |
| | \| | $x_1 = x_2(e)$ | function application |
| | \| | `if` $e$: $S_1$ else $S_2$ | conditional |
| | \| | `while` $e$: $S$ | while loop |
| | \| | $S_1$ | sequence |
| | | $S_2$ | |
| | \| | `return` $e$ | return |
| $e$ | ::= | $x$ | identifier |
| | \| | $n$ | number |
| | \| | $C\,(\overline{e})$ | object instantiation |
| | \| | $e\,.f\,(\overline{e})$ | method application |
| | \| | $[\overline{x}]$ | group |
| | \| | $e_1$.in($e_2$) | subgroup inclusion |
| | \| | $e_1$.union($e_2$) | union of groups |
| | \| | $e_1$.get($e_2$) | retrieve secret value from party $e_1$ |
| | \| | $e_1$.public_get($e_2$) | retrieve public value from party $e_1$ |
| | \| | `Secret`$\,(e)$ | result of $e$ becomes secret |
| | \| | `open`$\,(e_1,e_2)$ | result of $e_1$ becomes public for group $e_2$ |
| | \| | `openresult`$\,(e_1,e_2)$ | result of $e_1$ becomes public for group $e_2$ |
| | \| | `srand`$\,()$ | random secret number |
| | \| | `rand`$\,()$ | random public number |
| | \| | `lambda` $x$: $e$ | function expression |

Figure 3.3: Abstract syntax for Core PySMCL

$$
\begin{aligned}
&A = \texttt{Id}(C)\\
&B = \texttt{Id}(C)\\
&a = A.get(\ldots)\\
&b = B.get(\ldots)\\
&bigger = a > b\\
&open\_both = \texttt{open}(bigger, [A, B])
\end{aligned}
$$

Figure 3.4: Example program

## 3.7 Conclusion and Future Work

We have presented the current status on our preliminary work on the domain-specific language PySMCL embedded in Python. Our goal of PySMCL is to extend the security guarantees of SMCL to cover semantic security based on an explicit definition of the intended ideal functionality in the program.

We have described how we intend to embed PySMCL in Python in a seamless way by using function decorators and a carefull choice of domain-specific value constructors. The sketched implementation of PySMCL will be faithful to the dynamic nature of Python by not requiring a separate compilation/verification step as verification and preprocessing is done dynamically as the file is loaded by the Python interpreter

We also provided a core calculus for PySMCL which we expect will be a useful stepping stone for defining formal versions of the suggested analysis along with proofs of the security properties of PySMCL programs.

# Part II

# Towards a Strongly Typed Macro System

# Chapter 4

## Growing A Syntax

*Perfection is achieved not when there is nothing left to add,*
*but when there is nothing left to take away...*
— Antoine de St. Exupery, Wind, Sand, and Stars, 1939

Eric Allen  Ryan Culpepper  Janus Dam Nielsen

Jon Rafkind  Sukyoung Ryu

**Abstract**

In this paper we present a macro system for the Fortress programming language. Fortress is a new programming language designed for scientific and high-performance computing. Features include: implicit parallelism, transactions, and concrete syntax that emulates mathematical notation.

Fortress is intended to grow over time to accommodate the changing needs of its users. Our goal is to design and implement a macro system that allows for such growth. The main challenges are (1) to support extensions to a core syntax rich enough to emulate mathematical notation, (2) to support combinations of extensions from separately compiled macros, and (3) to allow new syntax that is indistinguishable from core language constructs. To emulate mathematical notation, Fortress syntax is specified as a parsing expression grammar (PEG), supporting unlimited lookahead. Macro definitions must be checked for well-formedness before they are expanded and macro uses must be well encapsulated (hygienic, composable, respecting referential transparency). Use sites must be parsed along with the rest of the program and expanded directly into abstract syntax trees. Syntax errors at use sites of a macro must refer to the unexpanded program at use sites, never to definition sites. Moreover, to allow for many common and important uses of macros, mutually recursive definitions should be supported.

Our design meets these challenges. The result is a flexible system that allows us not only to support new language extensions, but also to move many constructs of the core language into libraries. New grammar productions are tightly integrated with the Fortress parser, and use sites expand into core abstract syntax trees. Our implementation is integrated into the open-source Fortress reference interpreter.To our knowledge, ours is the first implementation of a modular hygienic macro system based on parsing expression grammars.

## 4.1 Introduction

A programming language can be thought of as a vocabulary of words and a set of rules that define how to combine words into meaningful constructs [55]. One goal of language design is to create a vocabulary and a set of rules that allow the programmer to express ideas clearly and concisely. But the set of concepts needed over the lifetime of a programming language is difficult to anticipate and is often dependent on the development of new systems that programs must interact with (for example, new domain-specific languages, new programming platforms, new virtual machines, etc.).

The Fortress programming language is intended to grow over time to accommodate the changing needs of its users [7]. One mechanism to allow for such growth is syntactic abstraction: It is possible to add new syntactic constructs to the language in libraries, defining new constructs in terms of old ones. In this manner, the language can gracefully adapt to unanticipated needs as they become apparent. Parsing of new constructs can be done alongside parsing of primitive constructs, allowing programmers to detect syntax errors in use sites of new constructs early. Programs in domain-specific languages can be embedded in Fortress programs and parsed along with their host programs. Moreover, the definition of many constructs that are traditionally defined as core language primitives (e.g. for loops) can be moved into Fortress' own libraries, thereby reducing the size of the core language.

Designing such a syntactic abstraction mechanism for Fortress is hard. In part, this difficulty is due to our design goal of growability: New syntax should be indistinguishable from old syntax from the user's perspective. This requirement imposes several constraints on Fortress macros: Macro definitions must be checked for well-formedness before they are expanded; otherwise syntax errors in the definition of a macro might not be exposed until the macro is used. Macro uses must be well encapsulated (hygienic, composable, respecting referential transparency); otherwise it would be impossible to safely use a macro without understanding the innards of its implementation. Use sites must be parsed along with the rest of the program and expanded directly into abstract syntax trees; otherwise new syntax would be distinguishable from old in the sense that use site errors of new syntax are not signaled alongside those of old syntax. Similarly, syntax errors at use sites of a macro must refer to the unexpanded program at use sites, never to definition sites. And because many desirable macro definitions require recursion, Fortress macros must support recursive and mutally recursive definitions.

Difficulties also arise from another design goal of Fortress: The concrete syntax of Fortress is designed to emulate mathematical notation as closely as possible. For example, operator precedence in Fortress is not transitive. Moreover, juxtaposition itself is treated as a mathematical operator, whose meaning is overloaded based on the types of the arguments. Juxtaposition of numeric variables denotes multiplication, as in the expression:

$$a(m + n)$$

Juxtaposition of a function with an argument denotes function application, as in the expression:

$$\sin x$$

It has been the experience of the Fortress design team that defining a grammar with these properties is most naturally done in the context of a formalism supporting unlimited lookahead. Fortress syntax is defined in the formalism of Parsing Expression Grammars (PEGs), which support unlimited lookahead, are unambiguous and are closed under union [41]. Consequently, any macro system for Fortress must work in the context of a core syntax defined as a PEG. To make the macro system as expressive

```
g1 = ⟨ 1, 2, 3, 4, 5 ⟩
g2 = ⟨ 6, 7, 8, 9, 10 ⟩
for i ← g1, j ← g2 do
  println "(" i ", " j ")"
end
```

```
g1.loop(fn i ⇒
  g2.loop(fn j ⇒
    println "(" i ", " j ")" ))
```

Figure 4.1: Left, a parallel for-loop in Fortress. Right, A desugared version of the for-loop on the left

as possible, we also allow new syntax to be defined via arbitrary PEGs. Thus, combination of new syntax with old involves composition of PEGs, and rules for resolving conflicts between separately defined syntax must be provided.

In this paper, we present a macro system for Fortress that meets these design challenges. Our system works in the context of a core syntax based on mathematical notation, and new syntax is indistinguishable from old. Indeed, we have found that constructs currently defined as part of Fortress core syntax can be moved to macro definitions in libraries. One such example is the Fortress `for` loop. In the left part of Figure 4.1, we define two variables $g_1$ and $g_2$, each referring to a list. Next, the `for` loop introduces the variables $i$ and $j$ which iterate through the two lists, and all pairs of values in the cross product of the two lists are printed. Our system allows `for` loops such as this to be transformed into method calls and anonymous functions as in the right part of Figure 4.1.

## 4.2 Contributions and Outline

The main contributions of this work are:

- Design of a hygienic macro system in the context of a core language based on parsing expression grammars
- A design for composing, and resolving conflicts among, separately compiled macros in the context of parsing expression grammars
- Explanation of the low-level mechanics of a working implementation of our system, available as open source in the Fortress reference implementation
- Definition of a core calculus for the Fortress macro system

The rest of the paper is organized as follows. Section 4.3 describes the design objectives of our system in detail. Section 4.4 describes how new syntactic abstractions can be described in the presented system through an example. Section 4.5 presents a formal treatment of the system semantics. In Section 4.6 the implementation of the system is presented and discussed. In Section 4.7 the macro system is evaluated by discussing various syntax extensions implemented in the system. A description of related work is given in Section 4.8, and we conclude in Section 4.9 along with discussing future work.

## 4.3 A Growable Language

A growable language is one of the key ideas of Fortress. The Fortress syntactic abstraction mechanism is designed to support the language growth with the goals described in this section in mind. Throughout this paper, we use the term *macro* to refer to a particular language extension.

**New syntax indistinguishable from the core syntax** It would be desirable if a macro syntax is indistinguishable from the core language syntax so that the uses of the macro do not introduce any visual clutter to the language. We could have chosen to enclose all macro uses in special brackets which would have made the parsing and recognition of macros easier. However, the result would have been a language in which extensions stand out conspicuously from core syntax, reminiscent of user-defined function definitions in APL [50]. 'Instead, we aim to design an extensible language with no apparent differences between the core syntax and macro uses, such as Scheme [4] and Lisp [54]. However, unlike the S-expressions of Scheme and Lisp, the Fortress core syntax emulates mathematical notation, which poses additional challenges in achieving this goal.

**Composition of independent macros** Composability of independent macros is especially important to support modular development of a system. Contributions from different development teams must be self-contained and must not interfere with each other. For example, suppose that one contribution provides a grammar with the following macro:

$$\text{Expr} \mid:= \ \mathit{macro}_1\ e : \text{Expr} \ \Rightarrow \langle\!\lceil e \rceil\!\rangle$$

which extends ($\mid:=$) an existing nonterminal Expr representing an expression with a new syntax "$\mathit{macro}_1\ e$" where $e$ is an expression, and translates ($\Rightarrow$) the new syntax to $e$. Now suppose that another contribution provides a grammar with the following macro:

$$\text{Expr} \mid:= \ \mathit{macro}_2\ e : \text{Expr} \ \Rightarrow \langle\!\lceil e^2 \rceil\!\rangle$$

which extends Expr with "$\mathit{macro}_2\ e$" where $e$ is an expression, and translates it to $e^2$. Then a grammar extending both grammars can use both macros as expressions, even in combination with each other:

$\mathit{macro}_1(\mathit{macro}_2\ \ 7)$
$\mathit{macro}_2(\mathit{macro}_1\ \ 7)$

**Expansion into other macros** Macros may be defined in terms of other helper macros. For example, the following macro:

$$\text{Expr} \mid:= \ \mathit{macro}_3\ e : \text{Expr} \ \Rightarrow \langle\!\lceil \mathit{macro}_4\ e \rceil\!\rangle$$

is defined in terms of the following helper macro:

$$\text{Expr} \mid:= \ \mathit{macro}_4\ e : \text{Expr} \ \Rightarrow \langle\!\lceil e \rceil\!\rangle$$

**Recursion and case dispatch** Many interesting macros can be nicely written using recursive applications of defined macros. The capability of recursively defining macros (including mutual recursion) and dynamically dispatching on macros improves expressiveness as we discuss in Section 4.8. For example, the following macro:

```
Expr|:= macro5 i : Id* do block : Expr end ⇒
        case i of
             Empty ⇒ ⟨⌈ block ⌉⟩
             Cons(first, rest) ⇒
                  ⟨⌈ do println first
                       macro5 rest do block end
                     end ⌉⟩
        end
```

defines a new syntax "$macro_5\ i$ `do` *block* `end`" where $i$ is a sequence of identifiers (Id$*$) and *block* is an expression, and its translation depends on the actual value bound to $i$. If $i$ is an empty sequence (Empty), the input sequence matched by the new syntax is translated to *block*. Otherwise, it prints the first identifier in $i$ (*first*) and recursively applies the new syntax to the remaining identifiers in $i$ (*rest*).

**Similar syntax for definition and use of a macro** If the syntax for defining a macro resembles the syntax for using the macro, it would be easy for a programmer to learn how to use the macro just by looking at its definition. Key features to enable this similarity between the definition site and use site of a macro include:

- Tokens appear in a macro definition where they appear in a use site of the macro, in contrast to other syntax specification languages [52, 83] where the terminal tokens are specified in a separate section.
- Whitespace characters between tokens are interpreted as optional Fortress whitespace characters, making the specification of a macro less bloated compared to using another character sequence to represent the specification of whitespace.

## 4.4 Syntactic Abstraction by Example

The Fortress syntactic abstraction mechanism provides a way to extend the core language syntax. Before formally describing the syntactic abstraction mechanism, we first provide some examples of how to define and use macros. Figure 4.2 presents a definition of a simplified syntax of `for` loops in Fortress.

```
grammar ForLoop extends { Expression, Identifier }
 Expr |:=
   for { i : Id ← e : Expr , ? Space }* do block : Expr end ⇒
   ◁[ for_2 i ∗ ∗; e ∗ ∗; do block; end ]▷
 | for_2 i : Id∗; e : Expr∗; do block : Expr; end ⇒
  case i of
    Empty ⇒
      ◁[ block ]▷
    Cons(ia, ib) ⇒
      case e of
        Empty ⇒◁[ throw Unreachable ]▷
        Cons(ea, eb) ⇒
          ◁[ ((ea).loop(fn ia ⇒ (for_2 ib ∗ ∗; eb ∗ ∗;
                                 do block; end)))]▷
    end
  end
end
```

Figure 4.2: A grammar definition of a simplified syntax of `for` loops in Fortress

A programmer can define a new syntax with a grammar definition. A grammar is a self-contained language extension which can be composed with other grammars in a modular way. An example grammar ForLoop is defined in Figure 4.2. A grammar contains a set of *nonterminal* definitions and extensions, and may extend any number

of other grammars, since it often makes sense to use macros from different grammars to implement new macros, or to create a new language extension by composing different languages. A new nonterminal is defined as:

NewNonterminal ::= ...

and an existing nonterminal is extended as:

ExistingNonterminal |:= ...

The available nonterminals in an extended grammar may be used as a basis for a nonterminal definition or extension. The ForLoop grammar extends an existing nonterminal Expr which defines all expression constructs in Fortress. It inherits the nonterminals provided by the extended grammars Expression and Identifier so that it can use the nonterminals in it. The Expr and Id nonterminals are defined in the Expression and Identifier grammars respectively and thus are available in ForLoop. We present the precise definition of nonterminal availability in Section 4.5.1.

A nonterminal definition or extension has an "ordered sequence" of alternative *variants*. For example, ForLoop extends Expr with two variants: the `for` loop variant and its helper variant $for_2$. A variant consists of a *pattern*, followed by $\Rightarrow$, and a *transformation expression*. A transformation expression is either an expression enclosed by $\lhd$ and $\rhd$, or a case expression with branches, whose right-hand sides are transformation expressions.

A pattern is defined by a sequence of parts. The pattern language is based on Parsing Expression Grammars (PEGs) and is described in Section 4.5. A part is either a simple part or operations over parts. Different kinds of parts make up the pattern for the `for` loop variant:

`for` { *i* : Id $\leftarrow$ *e* : Expr , ? Space }$*$ `do` *block* : Expr `end`

The first symbol is the terminal `for`, followed by a symbol group, another terminal `do`, a nonterminal reference "*block* : Expr", and then another terminal `end`. A terminal is any sequence of Unicode [95] characters that is not the name of a nonterminal available in the containing grammar. A symbol group is a sequence of symbols inside curly braces. A symbol or a symbol group may be followed by an option operator ? or a repetition operator + (one or more repetition) or $*$ (zero or more repetition). A nonterminal reference such as "*block* : Expr" is a reference to any nonterminal available in the containing grammar optionally prepended by a label followed by ":". The whitespace between symbols represents optional Fortress whitespace. When a space is required, the nonterminal Space specifies the required space. The pattern language supports unlimited lookahead in the same way as PEGs, by means of the semantic predicates: $\wedge$ (*must* match) and $\neg$ (must *not* match), both of which only match input but does not consume it.

A transformation expression gives meaning to the macro by providing a translation of the pattern. The transformation expression of the `for` loop variant:

$\lhd$ $for_2$ $i * *$; $e * *$; `do` *block*; `end` $\rhd$

uses the syntax-unfold operator, $**$, to expand the lists of identifiers and expressions as arguments to the $for_2$ variant. The syntax-unfold operator is similar to the ellipsis operator [84] found in Scheme [4], and expands into the syntax matched by its arguments. The $for_2$ variant is translated into core Fortress syntax by recursively traversing the identifier and expression lists. The transformation expression uses two nested case dispatches on the list structure of the pattern variables $i$ and $e$. The actual value bound to a pattern variable is compared to the left-hand side of each case clause: an empty list is matched to Empty and a nonempty list is matched to Cons. If a list is nonempty,

```
g1.loop(fn i ⇒
        for j ← g2 do
            println "(" i ", " j ")"
        end)
```

Figure 4.3: One unrolling of the for loop from Figure 4.1

the first argument of Cons contains the head of the list and the second argument contains the rest of the list. For each pair of head elements from an identifier list $i$ and an expression list $e$, we call the *loop* method on the head of the expression list, $ea$, with an anonymous function whose argument is the head of the identifier list, as the input to the *loop* method. The two lists of identifiers and expressions are guaranteed to have the same length by the structure of the `for` variant.

Although the transformation is expressed in terms of Fortress concrete syntax, the act of transforming a use site at parse time is not performed on the concrete syntax. Instead, the transformation expression is parsed into Fortress Abstract Syntax Tree (AST) nodes, containing *placeholder* nodes to represent pattern variables, and the act of transforming a use site is performed by substituting AST nodes in for the placeholders. An alternative approach would have been a multi-staged system in which transformation expressions consist of explicit user-defined computations of resulting AST nodes. Although such an approach is more flexible, it is also tedious and error prone. Moreover, it is difficult to check that syntax trees constructed through explicit computation are well-formed. Earlier work has shown that although a template-based approach is not as flexible as an approach based on explicit computation, it is usually possible to express the transformations desired in practice [20, 23].

The `for` loop example in Figure 4.2 shows that our syntactic abstraction mechanism satisfies the five goals described in Section 4.3:

- New syntax indistinguishable from the core syntax: As Figure 4.1 shows, the use of the `for` macro is indistinguishable from other Fortress language constructs.
- Composition of independent macros: The `for` macro extends the grammars, Expression and Identifier, so that it can use the nonterminals, Expr and Id, defined in them.
- Expansion into other macros: The `for` macro is defined in terms of the $for_2$ macro.
- Recursion and case dispatch: The $for_2$ macro is recursively defined using two nested case dispatches. Using the $for_2$ macro, the `for` loop in Figure 4.1 can be rewritten as in Figure 4.3. One more application of the macro will desugar the `for` loop as in the right part of Figure 4.1.
- Similar syntax for definition and use of a macro: The syntax for defining the $for_2$ macro:

  $for_2\ i : \text{Id}*; e : \text{Expr}*;$ `do` $block : \text{Expr};$ `end`

  and the use of it:

  $for_2\ i * *; e * *;$ `do` $block;$ `end`

  is very similar because tokens appear in the same place in both definition and use sites of the macro and implicit whitespace specification avoids visual clutter.

## 4.5 Syntax Normalization

In this section, we describe how a Fortress source program using macros is turned into a Fortress AST without macros which can be interpreted by the Fortress interpreter. For presentation brevity, we focus on a core subset of Fortress, *Core Fortress*. The abstract syntax of Core Fortress is presented in Figure 4.4.

A Core Fortress program consists of a possibly empty sequence of grammar definitions followed by an expression, called the *main expression*. Grammar definitions introduce new syntax to Core Fortress. A grammar definition includes a sequence of nonterminal definitions or extensions and it may extend other grammars to use nonterminals defined in the extended grammars in its definition. A nonterminal definition or extension has an ordered sequence of alternative variants and a variant consists of a pattern and a transformation expression (*Action* in Figure 4.4). A transformation expression is either a term enclosed by $\lhd$ and $\rhd$, called a *template*, or a `case` expression over a pattern variable. The nonterminals *GrammarName*, *NTName*, *PatternVar*, *Terminal*, and *Char* range over Unicode strings. An expression is a number, string, variable, function expression, function application, or parenthesized expression. A term is a pattern variable, an ellipses term, number, string, variable, function term, function application term, or parenthesized term. The metavariables $n$, $s$, and $x$ range over numbers, strings, and variables, respectively.

Because grammar definitions introduce new syntax to the language, the program itself defines how to parse it and turn it into an AST. We call the entire process from parsing a source program to creating a corresponding AST *syntax normalization*. Syntax normalization consists of two stages: *parsing* and *transformation*. In the parsing stage, the source program represented as a Unicode string is turned into what we call a *parsed* program. Then, in the transformation stage, the parsed program is transformed to a program in Core Fortress AST. Each stage is described in detail in the subsequent subsections.

| | | | |
|---|---|---|---|
| *Program* | ::= | *Grammar** *Expr* | |
| *Grammar* | ::= | `grammar` *GrammarName*<br>`extends` { *GrammarName** }<br>( *Definition* \| *Extension* )*<br>`end` | |
| *Definition* | ::= | *NTName* ::= *Variant** | |
| *Extension* | ::= | *NTName* \|:= *Variant** | |
| *Variant* | ::= | *Pattern* ⇒ *Action* | |
| *Pattern* | ::= | *Part** | |
| *Part* | ::= | *SimplePart*? | |
| | \| | *SimplePart*∗ | |
| | \| | *SimplePart*+ | |
| | \| | *SimplePart* | |
| | \| | ¬ *Part* | |
| | \| | ∧ *Part* | |
| *SimplePart* | ::= | *PatternVar*: *BasePart* | |
| | \| | *BasePart* | |
| *BasePart* | ::= | *NTName* | |
| | \| | *Terminal* | |
| | \| | [*Char*:*Char*] | |
| *Action* | ::= | ◁[*Term*]▷ | template |
| | \| | `case` *PatternVar* `of`<br>Empty ⇒ *Action*<br>Cons(*PatternVar*, *PatternVar*) ⇒ *Action*<br>`end` | |
| *Expr* | ::= | *n* | number |
| | \| | *s* | string |
| | \| | *x* | variable |
| | \| | `fn` *x* ⇒ *Expr* | function expression |
| | \| | *Expr Expr* | function application |
| | \| | ( *Expr* ) | parenthesized |
| *Term* | ::= | *PatternVar* | |
| | \| | *Term* ∗∗ | |
| | \| | *n* | |
| | \| | *s* | |
| | \| | *x* | |
| | \| | `fn` *x* ⇒ *Term* | |
| | \| | *Term Term* | |
| | \| | ( *Term* ) | |

Figure 4.4: Abstract syntax for Core Fortress

### 4.5.1 Parsing

The parsing stage transforms a source program in a Unicode string into a parsed program in an AST representation which describes how the program is to be transformed into an executable Core Fortress program. The transformation is nontrivial because a program itself defines how it is parsed: the grammar definitions in the program determines which syntax may occur in the grammar definitions and the main expression.

In order to resolve this self-dependency problem, we take a two-step approach to parse Fortress programs. In the first step, we parse all the grammars except for the action part of each variant and the main expression. The action parts and the main expression are just parsed as Unicode strings. This step is a standard parsing which generates an AST where expressions are represented as strings. In the second step, we parse each action part and the main expression. In order to parse the action parts, we need to compute what we call the set of *available macros*. In a later subsection, we show how to compute the set of available macros for a given grammar, and how to derive a PEG from which a PackRat parser can be generated (for example by using Rats! [46, 47]) and then used to parse the action parts. The action parts and the main expression are parsed into *node expressions*. A node expression describes how macros are invoked to turn a macro syntax into a Core Fortress syntax. The construction of node expressions is described below. This two-step approach works because the grammar definitions appear before the main expression. The approach also has the advantage that we break the dependency in parsing recursive macros.

The parsing stage relies on *Parsing Expression Grammars* (PEGs) [41] because they have some advantages over usual Context-Free Grammars (CFG) [26] based parsing formalisms such as LL and LALR(k). PEGs are unambiguous, are closed under union, and integrates lexing with parsing. We need the closure property because we want to combine PEGs for different grammars and the integrated lexing and parsing makes it easy to achieve our goal of "similar syntax for definition and use". Furthermore, the parsers based on PEGs allow a linear execution time compared to the generalized CFG parsers. We briefly introduce PEGs in the next subsection along with a description of our pattern language which is effectively a variant of *Parsing Expressions*.

#### Parsing Expression Grammars

A Parsing Expression Grammar (PEG) is a 3-tuple $(N, T, s)$, where $N$ is a finite set of nonterminal definitions, $T$ is a finite set of terminal symbols, $s \in N$ is the start nonterminal definition. A nonterminal definition is a pair $(n, cs)$ where $n$ is a name and $cs$ is a list of prioritized alternatives, each alternative being a *parsing expression* [41]. The terminal symbols and the nonterminal names are disjoint. The empty string $\epsilon$, a terminal symbol $a \in T$, and a nonterminal name $n$ are parsing expressions, and if $e, e_1$, and $e_2$ are parsing expressions then so are a sequence $e_1\ e_2$, an optional expression $e?$, zero-or-more repetitions $e^*$, a not-predicate $\neg e$, and a and-predicate $\wedge e$.

The main difference between PEGs and CFGs is the lack of ambiguity in PEGs resulting from the prioritized alternatives. In a CFG, the two nonterminal definitions $A = a \mid ab$ and $A = ab \mid a$ are equivalent. However, in a PEG, the second alternative of the former would never succeed because the first alternative is always taken if the input string to be recognized starts with $a$.

The pattern language shown in Figure 4.4 is based on parsing expressions with some differences. If $Part_i$ $(1 \leq i \leq n)$ of a sequence of parts, ($Part_1$, $Part_2$, ..., $Part_n$), corresponds to a parsing expression $e_i$, then the sequence of parts corresponds to the sequence of parsing expressions with optional Fortress whitespace in between: ($e_1$, $w$, $e_2$, $w$, ..., $w$, $e_n$). Here $w$ refers to the nonterminal defining the optional whitespace in Fortress. This small difference reduces the visual clutter when writing

```
grammar A
  Nt::=
      macroA ⇒ ...
end

grammar B extends {A}
  Nt|:=
      macroB ⇒ <|... macroA ...|>
end

grammar C extends {A}
  Nt|:=
      macroC ⇒ ...
end
```

Figure 4.5: Grammars including multiple nonterminal extensions

language extensions. We also allow character classes to be specified (e.g. the character class of ";" and the lowercase letters from "a" to "z" [;*a*:*z*]). Character classes can be trivially, but tediously, encoded in parsing expressions by explicit enumeration of the alternatives.

### Combining Grammars

In order to create a parser and parse the action parts in a given grammar, we need to compute the set of available macros. The set of available macro definitions for a grammar consists of the macros defined by the nonterminal definitions and extensions in the grammar and inherited from the extended grammars. However, the multiple "inheritance" of grammars in combination with the prioritized alternatives of PEGs makes it nontrivial to compose nonterminal extensions.

For example, Figure 4.5 shows three grammars $A$, $B$, and $C$, each of which defines a single macro. We write $\lhd|\ldots macroA\ldots|\rhd$ to indicate a template where the macro *macroA* occurs somewhere. Grammar $B$ and $C$ extend grammar $A$ which contains the definition of the nonterminal Nt.

```
grammar D extends {B}
  Nt|:=
      macroD ⇒ <|... macroB ...|>
    | B.Nt

(* This definition is illegal,
    because macroA is not propagated by the grammar B:
  Nt|:=
      macroD2 ⇒ <|... macroB ... macroA ...|>
*)
end
```

Figure 4.6: Grammars should behave as usual module systems

The interesting question is what the semantics of a nonterminal extension should be. It seems natural to expect that $B$ is able to use all the macros defined in $A$ since it extends it. However, what if a grammar such as $D$ in Figure 4.6 extends $B$? Because $D$ does not extend $A$ directly, we do not allow *macroA* to be available in the grammar

```
grammar E extends {B, C}
  Nt|:=
      macroE ⇒ ⊲[. . . macroB . . . macroC . . .]⊳
end
```

Figure 4.7: Multiple extensions of a single nonterminal

$D$ as the comment enclosed by "(*" and "*)" in Figure 4.6 shows. This behavior is consistent with the Fortress component system where a component does not reexport all of its imports by default. A grammar may explicitly propagate an inherited macro by referring to the nonterminal extension defining the macro. The second alternative of the nonterminal extension in $D$ gives an example of explicit propagation of an inherited macro: $B$.Nt. Whenever *macroD* is available, *macroB* is also available.

Grammar $B$ overrides the definition of the Nt nonterminal in grammar $A$, but the override does not affect grammar $C$ which also extends $A$. It is only possible to completely override a nonterminal if all uses of the grammar containing the overridden nonterminal is changed to the grammar containing the overriding nonterminal. In the example above $C$ would have to extend $B$ instead, and there is no guarantee that another grammar which extends $A$ could be created.

A grammar may extend multiple grammars. Multiple extension induces a lattice structure in the extension relation. One key issue with multiple extension is how to combine grammars when there is a "diamond" shape in the lattice as shown in Figure 4.7. The grammar $E$ extends $B$ and $C$ both of which extend the same grammar $A$ but define different extensions for the same nonterminal Nt. One approach would be to collect the macros along the paths from $B$ and $C$ to $A$ following the extension relation, and then concatenate them:

```
Nt|:=
    macroE ⇒ ⊲[. . . macroB . . . macroC . . .]⊳
  | macroB ⇒ ⊲[. . . macroA . . .]⊳
  | macroA ⇒ . . .
  | macroC ⇒ . . .
  | macroA ⇒ . . .
```

The order of *macroB* and *macroC* could be reversed, but it has the same problem as this approach: the syntax defined in $A$ might shadow *macroC* because the variants are ordered. It is an unfortunate behavior because we intend to support a user-defined syntax on top of the existing Fortress syntax. Both *macroB* and *macroC* should have higher priority than *macroA*.

Our solution to the shadowing problem is to collect the macros along the paths till the lowest common ancestor of the immediately extended grammars in the extension relation and concatenate them, and then recursively collect and concatenate the macros along the paths to the end of the extension relation. For example, we resolve the nonterminal Nt defined in grammar $E$ as follows:

```
Nt|:=
    macroE ⇒ ⊲[. . . macroB . . . macroC . . .]⊳
  | macroB ⇒ ⊲[. . . macroA . . .]⊳
  | macroC ⇒ . . .
  | macroA ⇒ . . .
```

A thing to note is that the ordering of macros must be specified. The ordering should be deterministic and preferably the grammar writer should be able to control it. Therefore,

$$
\begin{array}{lcl}
\mathit{NodeExpr} & ::= & \mathit{PatternVar} \\
& | & \texttt{case } \mathit{PatternVar} \texttt{ of} \\
& & \quad \text{Empty} \Rightarrow \mathit{NodeExpr} \\
& & \quad \text{Cons}(\mathit{PatternVar},\ \mathit{PatternVar}) \Rightarrow \mathit{NodeExpr} \\
& & \texttt{end} \\
& | & \mathit{Transformer}\ (\overline{\mathit{NodeExpr}}) \\
& | & \text{Ellipses}\ (\mathit{NodeExpr}, \overline{\mathit{NodeExpr}}) \\
& | & \mathit{NodeConstructor}(\overline{\mathit{NodeExpr}})
\end{array}
$$

Figure 4.8: Abstract syntax for node expressions

we have chosen to use the order in which the extended grammars appear in the `extends` clause.

Using the solution, we build a PEG from which a parser can be generated and used to parse the action parts and the main expression. The PEG for a grammar is:

- formed from all definitions from all transitively-extended grammars and the current grammar,
- modified by the extensions from the current grammar, if any, and
- modified by the implicit extensions for all nonterminals in the inherited extensions that are not extended in the current grammar.

The first step of constructing the PEG is to gather all of the nonterminal definitions from all the transitively-extended grammars. A variant in a nonterminal definition is either a pair of a syntax pattern and a transformer or a reference to an extended grammar's extension (propagation of an extension). For the former, the pair is simply added to the PEG. For the latter, we retrieve the extensions added by that grammar, process them, and add them to the PEG. The next step is to apply the extensions. The extensions consist of the explicit extensions (defined in the grammar) and the implicit extensions inherited from the extended grammars. The PEG for parsing the main expression is obtained in the same way as for a grammar which extends all the preceding grammars, but containing no definitions nor explicit extensions.

### Construction of Node Expressions

Using the parser generated by a PEG as described in the previous subsection, the action parts in the grammars and the main expression are parsed into node expressions. A node expression is a representation of how macro invocations are applied to obtain a Core Fortress AST. The abstract syntax of node expressions is presented in Figure 4.8 where a sequence of *NodeExpr* is represented as $\overline{\mathit{NodeExpr}}$. A node expression is a pattern variable, a case dispatch, a macro invocation represented as a reference to a *Transformer* applied to a number of node expression arguments, an ellipses node represented as a reference to an Ellipses applied to a node expression (the first argument), or a Fortress expression represented as a reference to a *NodeConstructor* applied to a number of node expression arguments. We use *Transformer* and *NodeConstructor* to range over templates and Core Fortress expressions, respectively. The second argument of the Ellipses node is initialized with its first argument and changes during the evaluation of the node.

A node constructor is created when an input string is matched by the Core Fortress syntax. A transformer is created whenever a macro use is parsed by matching the macro definition against the macro use. A macro definition is matched in the same way as

a parsing expression is matched against an input string: each part is matched in the order it appears. If the macro definition contains any references to other nonterminals, then we parse with respect to the nonterminals and if they succeed the result is a node expression, and we continue parsing the rest of the macro definition. If all the parts in a macro definition are matched, then a transformer is created with the node expressions from any nonterminals as arguments.

Consider an example from Section 4.3, where $macro_1$ is applied to the literal 7:

$$macro_1 \ \ 7$$

and parsed against the macro definition:

$$\text{Expr} \mid:= \ macro_1 \ e : \text{Expr} \ \Rightarrow \langle [e] \rangle$$

First, we see that there are two parts in the macro definition: $macro_1$ and "$e : Expr$". The first part is a terminal part which must be matched verbatim in the input sequence and the other part is a binding part where the pattern variable $e$ gets bound to the result of parsing *Expr*. Parsing the action part of $macro_1$ returns the node expression consisting of a pattern variable reference and is transformed to a *Transformer* $T_1$, where $T_1$ is a fresh name. Then, parsing the macro use requires parsing 7 against the *Expr* nonterminal, which is matched by the Core Fortress syntax and the result is a node constructor for the Core Fortress abstract syntax $7$. We bind $e$ to $7$ in the environment and pass $7$ as an argument to the transformer $T_1$, resulting in the node expression $T_1(7)$ which can be passed on to the transformation phase described in Section 4.5.2.

### 4.5.2 Transformation

Transformation is the evaluation of node expressions to construct a Core Fortress AST without macros which can be interpreted by the Fortress interpreter. Given a parsed program which contains a set of grammars and a main expression, we transform it to an executable Core Fortress program by evaluating the node expression first in the main expression and consequently in transformers.

We define the evaluation of node expressions in terms of a small-step operational semantics [74]. We define the semantics using a transition system. The transition system consists of a set of configurations $\sigma \in \Sigma$, a set of terminal configurations $\mathrm{T} \subseteq \Sigma$, and a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$. A configuration is a node expression along with two environments: $\Upsilon, \Gamma \vdash NodeExpr$. The transformer environment $\Upsilon$ maps transformer names to transformer definitions and the node environment $\Gamma$ maps pattern variables to node expressions. We write "$\Gamma \left[\overline{x} \mapsto \overline{n'}\right]$" to denote an extension of $\Gamma$ with the new bindings "$\overline{x} \mapsto \overline{n'}$". The transformer environment is initialized to contain all the transformers from all the grammars and does not change during evaluation. This is possible because the parsing has already resolved the scope of the transformers introduced by the grammars. The node environment is initially empty. A value $v$ is a node constructor application, the arguments of which are also values or a list of values.

The evaluation of a node expression is inductively defined by the transition relations in Figure 4.9. The metavariables $t$ ranges over transformer names, $x$ ranges over pattern variables, $n$ ranges over node expressions, and $c$ ranges over node constructors. For brevity, we write "$t\,\overline{x}.n$" for a transformer definition of name $t$, which takes the pattern variables $\overline{x}$ and has the body $n$. If the node expression is a pattern variable, we look it up in the node environment. If the node expression is a case-dispatch, we first evaluate the condition $x$ in the current environments. The result is either an empty list Empty or a nonempty list $\mathrm{Cons}(hd, tl)$ with the first element $hd$ and the rest of the elements $tl$. If it is an empty list, we evaluate $n_1$ in the current environments. If it is a nonempty list, we extend the node environment with the bindings for the

pattern variables $x_1$ and $x_2$ to *hd* and *tl* respectively, and continue the evaluation. If the node expression is a macro invocation, we look up the transformer definition in the transformer environment, evaluate the macro arguments, and extend the node environment with the bindings from the pattern variables to their actual values, and continue evaluating the body. If the node expression is an ellipses node, the first argument of the node is replicated by the number of times equal to the length of a pattern variable within the first argument. The second argument of the ellipses node keeps track of the intermediate results during the evaluation of the ellipses node. For brevity, we write "$PV(n)$" for a set of pattern variables in the node expression $n$, "$l.nth(i)$" for the $i$-th element of the list $l$, and "$size_\Gamma(n)$" for the length of the value bound to any pattern variable within the node $n$ in the environment $\Gamma$. If no value is bound to any pattern variable, $size_\Gamma(n)$ returns 0. If the node expression is a node constructor invocations, we construct the corresponding Core Fortress node.

The evaluation of a node expression may either fail, diverge, or result in an executable Core Fortress node. The evaluation may diverge if there is an infinite recursion in the definition of a transformer.[1] We do not consider nontermination a problem, because it will mainly happen when a macro does not deconstruct input, and programmers writing programs with recursive functions are used to deal with this kind of problems.

### 4.5.3 Hygiene

Fortress macros are hygienic, meaning any bindings introduced by a macro are fresh. Our hygienic system is a simplification of Clinger's algorithm [28]. Since our macro system cannot introduce macros itself, nor can a macro be hidden through a lexically bound variable, we need not concern ourselves with much of the complexity of the original hygiene algorithm.

The hygienic transformation works as follows. Before a macro is invoked, no renaming is performed. After a macro invocation, a flag is set which indicates that renaming should take place for the transformation of the current node and all the children nodes. During hygienic transformation, identifiers are looked up in a syntactic environment and replaced with the renamed identifiers they are bound to. The initial syntactic environment is the identity environment. Once the flag indicating that renaming should occur is set, each language construct binding an identifier generates a unique identifier and maps the original bound identifier to the new unique identifier. Each binding construct introduces a new syntactic environment which maintains a link to the previous environment. Identifiers not found in the immediate environment are recursively looked up in the parent environment until a mapping is found.

Pattern variables in a macro definition are never renamed since they are syntactic entities being introduced into the macro. In this way, we prevent variables passed to the macro from being renamed by virtue of the fact that they share names with bound identifiers introduced by the macro. Consider the following macro definition:

$$\text{Expr}|:= \mathit{foo}\ e : \text{Expr} \Rightarrow \lhd [\, \texttt{fn}\ d \Rightarrow d + e \,] \rhd$$

If the macro is invoked as:

$$\mathit{foo}\ d$$

then the result after hygienic transformation should be:

$$\texttt{fn}\ d_1 \Rightarrow d_1 + d$$

[1] Transformation described in this section is similar to Wand's [58] algorithm. The transformation function implements $\tau$ while the parser implements $\beta$ and D.

[Pattern Variable]

$$\frac{\Gamma(x) = v}{\Upsilon, \Gamma \vdash x \to \Upsilon, \Gamma \vdash v}$$

[Case Empty]

$$\frac{\Upsilon, \Gamma \vdash x \to \Upsilon, \Gamma \vdash \text{Empty}}{\begin{array}{l}\Upsilon, \Gamma \vdash \texttt{case}\ x\ \texttt{of} \qquad\qquad \to \Upsilon, \Gamma \vdash n_1 \\ \quad \text{Empty} \Rightarrow\ n_1 \\ \quad \text{Cons}(x_1, x_2)\ \Rightarrow\ n_2 \\ \texttt{end}\end{array}}$$

[Case Cons]

$$\frac{\Upsilon, \Gamma \vdash x \to \Upsilon, \Gamma \vdash \text{Cons}(hd, tl)}{\begin{array}{l}\Upsilon, \Gamma \vdash \texttt{case}\ x\ \texttt{of} \qquad \to \Upsilon, \Gamma\,[x_1 \mapsto hd]\,[x_2 \mapsto tl] \vdash n_2 \\ \quad \text{Empty} \Rightarrow\ n_1 \\ \quad \text{Cons}(x_1, x_2)\ \Rightarrow\ n_2 \\ \texttt{end}\end{array}}$$

[Macro Invocation Arguments]

$$\frac{\Upsilon, \Gamma \vdash \overline{n} \to \Upsilon, \Gamma \vdash \overline{n'}}{\Upsilon, \Gamma \vdash t(\overline{n}) \to \Upsilon, \Gamma \vdash t(\overline{n'})}$$

[Macro Invocation]

$$\frac{\Upsilon(t) = t\,\overline{x}.n}{\Upsilon, \Gamma \vdash t(\overline{v}) \to \Upsilon, \Gamma\,[\overline{x} \mapsto \overline{v}] \vdash n}$$

[Ellipses First]

$$\frac{\begin{array}{c}|\overline{v}| + 1 = i \leq \mathit{size}_\Gamma(n) \\ \Gamma' = \mathit{extend}_\Gamma(PV(n), i) \qquad \Upsilon, \Gamma' \vdash n \to \Upsilon, \Gamma' \vdash n'\end{array}}{\Upsilon, \Gamma \vdash \text{Ellipses}(n, \langle \overline{v} \rangle) \to \Upsilon, \Gamma \vdash \text{Ellipses}(n, \langle \overline{v} n' \rangle)}$$

[Ellipses Last]

$$\frac{|\overline{v}| = \mathit{size}_\Gamma(n)}{\Upsilon, \Gamma \vdash \text{Ellipses}(n, \langle \overline{v} \rangle) \to \Upsilon, \Gamma \vdash \overline{v}}$$

[Node Constructor]

$$\frac{\Upsilon, \Gamma \vdash \overline{n} \to \Upsilon, \Gamma \vdash \overline{n'}}{\Upsilon, \Gamma \vdash c(\overline{n}) \to \Upsilon, \Gamma \vdash c(\overline{n'})}$$

where $\mathit{size}_\Gamma(n) = \mathit{max}(|\Gamma(x_j)|) \quad \forall x_j \in PV(n)$

$$\mathit{extend}_\Gamma(\{\overline{x_j}\}, i) = \begin{cases} x_j \mapsto \Gamma(x_j) & \text{if } |\Gamma(x_j)| = 1 \\ x_j \mapsto \Gamma(x_j).\mathit{nth}(i) & \text{if } |\Gamma(x_j)| > 1 \end{cases}$$

Figure 4.9: Operational semantics for node expressions

## 4.6 Implementation

The syntactic abstraction mechanism is built on top of Rats!, the underlying parser technology used to parse Fortress programs. A key aspect of Rats! is its composition framework based on modules. By separating grammar extensions in their own module, we were able to extend the core Fortress grammar in a modular way.

A new parser is generated for each modified or defined nonterminal in an extension grammar. All the nonterminals in the grammar are compiled into Rats! syntax and added into a new module, $\texttt{M}_{\text{user}}$. A nonterminal that is extended originally has productions $p_1 \ldots p_n$ and is modified so that $\texttt{M}_{\text{user}}$.`nonterminal` is inserted before $p_1$. All productions that refered to `nonterminal` will now implicitly refer to $\texttt{M}_{\text{user}}$.

The macro system is logically broken up into two stages: parsing, and transformation. The *first* stage consists of parsing the grammar declaration into an AST which contains transformation nodes instead of the template bodies. This stage has two steps: parsing the grammar declaration and parsing the template bodies. The core fortress parser is used to parse the grammar declarations and then those results are used to construct a new parser that is used to parse the templates. The system constructs a new parser for each unparsed template body and replaces the grammar AST node with a transformation node which contains the parsed template body. When a component is parsed, the grammars that it imports define the PEG used to parse the component body. The resulting AST will have nodes that specify a macro invocation. In stage *two*(transformation) the AST will be run through a macro processing engine which converts macro invocation nodes into their corresponding template bodies.

Template bodies consist of regular Fortress syntax, macro invocations, pattern variables, and ellipses nodes. During transformation, the macro system will dispatch according to these four cases. For regular Fortress syntax the engine will recursively dispatch on child nodes but otherwise do no additional processing unless a macro has already been invoked in which case the hygiene rules will be applied. Macro invocations will cause the engine to look up the function defined for that macro and apply it to the parameters of the macro. Pattern variables are looked up in the current macro's environment and replaced with a corresponding expression. Ellipses nodes cause the engine to replicate the immediate child node of the ellipses node a number of times equal to the length of a pattern variable that is also inside the ellipses node. The replicated nodes are then spliced into the parent node.

## 4.7 Evaluation

We have evaluated the design of our syntactic abstraction system by implementing the `for` loop example shown in Figure 4.2, and also a grammar that recognizes regular expressions and one recognizing XML. These examples show that the system is suitable for language extensions and domain specific languages by allowing their care-free implementation in general. The grammars for XML and regular expressions can be found as part of the open-source Fortress interpreter [6].

The for loop example demonstrates a weakness of the macro system in that some macros must be split up to accomodate the parser. The for loop cannot be written directly because when the ellipses are expanded the intermediate commas are missing and so the parser would not recognize the syntax as a for loop macro. Instead, the body of the for loop is written in such a way that no syntactic markers are needed and a separate macro is invoked with this simpler body. This generally means that most macros will need two forms and macro writers will have to know about both forms when invoking another macro inside their template. A solution to this problem would be to add operations over lists, so that the identifiers and the corresponding expressions

in the for loop example could be zipped, and expanded with the commas in place. In general it would be useful if one could mark local helper macros as such, e.g. by using visibility modifiers like `private` in Java.

The system has a number of limitations. Syntax extensions are imported at the component level, and so we do not support lexically scoped syntax extensions like Scheme [4] or OMeta [105] does. Support for lexical scoping could be added later. A number of other systems [47, 105] based on PEGs allow semantic predicates in parsing expressions. A semantic action is a boolean-valued expression which must be true in order for a parsing expression to match. Semantic actions are useful in some situations, but in our exprience we haven't had any need for semantic actions in any of our examples yet.

Currently the specification of syntax is not separated from the specification of semantics(actions), e.g. like in Bracha's [21]. Such a separation can be useful, e.g. for visualizing the source code in IDEs, or supporting different transformations. Explicit separation of the syntax and transformation is possible and we would like to investigate this in the future.

The current implementation uses Rats! to generate a parser for each transformation expression. The overall parsing of a Fortress program performs comparable to the parsers generated by Rats!, that is linear time. However there is the overhead of generating a grammar and invoking Rats! on that grammar for each transformation expression, which currently slows down the parsing first time a macro is used. The following times are for free because the generated parser is cached. The size of the transformed Core Fortress program is in worst case exponential in the size of the input.

Much of the syntactic abstraction system functionality falls out from the mechanisms inspired by Scheme systems, however the uniform and elegant structure of s-expressions cannot be reused in the context of Fortress, due to the many nuances of concrete syntax of Fortress. This prevented us from writing procedural macros efficiently, the amount of code required to construct a Fortress AST makes them an unattractive path to take.

## 4.8 Related work

Our system for syntactic abstraction incorporates ideas from two main lines of research: extensible grammars for meta-programming; and Lisp and Scheme macro systems.

Meta-programming systems and extensible grammar systems provide frameworks for constructing languages, extending languages with domain-specific notations, and translating between languages. Typically, a meta-program manipulates a separate object-program. Sometimes the meta-program and object-program are written in different languages. In contrast, Lisp and Scheme macros affect subsequent parts of the same program that contains the macros. Using macros, programmers can write syntax extensions and domain-specific languages as libraries, with no need for preprocessors or custom compilers. Fortress takes this approach of supporting extension from within, rather than making it a separate facility with separate tools.

### 4.8.1 Extensible Syntax and Meta-programming

There are numerous languages, libraries, and frameworks designed for creating language tools, from simple parser generators [52] to comprehensive language definition frameworks. In this section we discuss a few systems that offer special support for language extension or translation. Rats! [46, 47] is a tool for producing packrat [41] parsers from modular grammars. Using the module system a programmer can create

new languages by applying modifications to existing grammars. Rats! provides parsing support for Fortress syntactic abstraction system.

OMeta [106] is an embedded language (in COLA [29] or Squeak Smalltalk [90]) which combines pattern matching and parsing. OMeta is based on PEGs which are generalized to work on arbitrary datatypes not only streams of characters, which leads to the use of PEGs for pattern matching. The encapsulation of grammars in OMeta is also inspired by object-oriented constructs but in contrast to grammars in Fortress they are encapsulated in a "class" like construct which only supports extension of a single grammar, the reason is mainly for avoiding nameclashes. Single extension is not a good match for expressing grammar extension since it is often the case that a grammar extends multiple other grammars. This is solved in OMeta though the use of the "foreign" production which effectively dispatches to separate grammars. Fortress solves this issue by allowing a grammar to extend multiple other grammars. "foreign" in Ometa also does not allow the programmer to selectively provide nonterminals to the user.

Katahdin [85] is a platform for specification and implementation of programming languages. A language is specified using a language where both the syntax and semantics are mutable at runtime. Katahdin is based on PEGs with some modifications, the choice is not prioritized but ordering is resolved based on a longest match strategy. Katahdin allows whitespace everywhere unless not explicitly disallowed, similar to our approach. However they go a step further and allow the programmer to redefined whitespace locally. A similar approach would benefit the Fortress syntactic abstraction system.

MetaBorg [22,23,77,100] is a method of adding domain-specific notation support to general-purpose languages. The method relies on a syntax definition framework called SDF [48]. SDF permits definitions of arbitrary context-free grammars, which enables modular development of syntax. Another component in the MetaBorg tool chain is a term-rewriting language called Stratego. SDF and Stratego together support the construction of program transformers; SDF is used to describe the syntax and embed it into a Stratego program that specifies the transformation rules. The MetaBorg tools support language extension, but they constitute an external mechanism rather than an internal, integrated extension mechanism.

Silver [107] is a language description framework based on attribute grammars designed for the modular construction and composition of domain-specific languages. The semantics of the language is defined by its nonterminals' attributes. Silver supports a feature called forwarding [108] that allows the semantics of one syntactic form to be defined in terms of a translation into another syntactic form.

Cardelli et al. [25] devised an extensible syntax system based on grammar modifications. Like Lisp and Scheme but unlike most meta-programming systems, their system allowed programs to manipulate their own syntax; a programmer could introduce a new form and then use it in the same program. Translations in their system were given in terms of concrete syntax patterns, allowing syntactic abstractions to be built incrementally. Like Scheme's hygienic macro systems, their system also preserves lexical scoping via renaming, but they have an explicit mechanism for requesting a fresh variable name. Their system is more restrictive than ours in several ways. First, they use an LL(1) parser, and consequently their parser is efficient, but they cannot handle many of the syntaxes supported by Fortress. Their system performs transformations as part of the parsing process, and translations are analyzed to assure the termination of parsing. In contrast, our system has a separate translation pass, and while parsing is guaranteed to terminate, the translation pass might not.

### 4.8.2 Lisp and Scheme Macros

Fortress borrows several ideas from Lisp and Scheme macros. Fortress syntax extensions are part of Fortress programs, not external metadata requiring special handling. We also take heed of Scheme's consideration for proper lexical scoping. Finally, the semantics of grammars is designed to support the same kind of modularity patterns as Scheme's module systems do.

In 1986, Kohlbecker et al. [57] introduced *hygienic* macro expansion to prevent the inadvertent variable captures that sometimes occur in naive macro expansion. The original hygiene algorithm handled only top-level macros. Researchers in the Scheme community extended hygiene to cover locally defined macros [28, 39], first-order modules [40,89,104], and dynamically-linked components with macros in the signatures [33]. The original algorithm worked with top-level macros implemented as Lisp functions on syntax trees, and it involved scanning the result of every macro call to rename newly introduced identifiers. The algorithm developed by Clinger and Rees [28] eliminated the repeated code scanning by restricting macros to use fixed templates rather than arbitrary code to compute their results.

Our syntactic abstraction system implements hygiene in a manner that combines features of the Kohlbecker algorithm and the Clinger and Rees algorithm. Fortress macros occur only in grammars defined in APIs, so we do not need the machinery for locally defined macros, and Fortress macros specify their translations in terms of templates, so we do not need deep code scans.

The Fortress grammar system is designed to support modular syntax extensions. A common pattern in Scheme is to define one macro in terms of several auxiliary macros but only export the main macro. In our system, a syntax extension can be defined with the help of other extensions, but clients of the one extension are not forced to accept the auxiliary extensions. In Scheme, this is done through the scoping of the macro name; in our system, grammars control the extent of changes to nonterminals.

Finally, the entire syntactic abstraction system resides ultimately in the APIs. Components are separately compiled; a component can be compiled given only the relevant APIs. Syntactic abstractions can refer to types, variables, and functions declared in the API, and the references they produce in client components are resolved based on the components the client is linked to. The Scheme component system with macros in the signatures [33] is arranged the same way and has similar hygiene properties.

## 4.9 Conclusion and Future Work

A growable language is one of the main ideas of the Fortress programming language. The Fortress syntactic abstraction mechanism serves a key role to support the language growth. It allows a user-defined syntax to be indistinguishable from core Fortress syntax, provides composition of independent macros, and supports mutually recursive macros using dynamic dispatches. The Fortress syntactic abstraction mechanism is so flexible that various language constructs can be moved from the core language syntax into libraries using macros. Type checking macros is an interesting future direction. Because the syntactic abstraction mechanism described in this paper is a template-based approach, it is possible to perform a more precise type checking than a multi-staged system where the transformation is represented as an explicit construction of the AST nodes. The goal of type checking macros would be to provide the static guarantee that if a macro definition is well typed then there is no type error at the use sites of the macro unless the user of the macro provides an input of a wrong type.

The authors sincerely thank the Fortress team for fruitful discussions and support. Thanks to Robert Grimm for answering our questions about Rats!, and thanks to the anonymous reviewers for suggestions on improving the paper.

# Chapter 5

## Towards Strongly Typed Macros

*Good fences make good neighbors*
— Robert Frost, "Mending Wall"

### 5.1 Introduction

In this Chapter we present a description of our preliminary work on a strongly typed macro system based on the work presented in Chapter 4.

Macros written in the macro system described in Chapter 4 may contain type errors, because no type checking occurs on the macro definitions. Type errors may manifest themselves in the expanded code and thus become a very surprising experience for the macro user. Especially if the error happens deeply inside the body of the macro, and does not depend on the macro arguments in a trivial way. Furthermore there is no other way to correct the type error but to modify the macro definition which can be difficult especially if the user did not write the macro himself, or even impossible if the code is not available. This situation is unsatisfactory because the host language, Fortress [7], is a strongly typed language.

We want to develop a modular type checking of macro definitions by requiring the macro definer to annotate macro definitions with types, and then have the type checker check that the macro definition is type correct under the given annotations. In this way, we can ensure that all type errors in a macro definition are found at the declaration site, and that the only type errors that can occur at a use site are those resulting from the user providing arguments of the wrong types.

As an example we look at the implementation of `letcc` shown in the left grammar of Figure 5.1, where the `letcc` macro takes two arguments; the identifier $k$ and the expression $e$. The macro is desugared to a label expression, a function definition, and the expression $e$, where we use the identifier $k$ as the name of the function.

We would like to type check such macros, in order to do so we need to assess what information is needed. In the `letcc` example we clearly see that expressions provided as arguments, like $e$, must be given a type. Therefore it is reasonable to require $e$ to have some type T, where T is a type variable to allow for generic use of `letcc`. We will require the macro definer to provide these types as annotations, but we hope that we in the future will be able to infer those types, to lighten the burden of the programmer. We suggest that the annotation is placed after the expression $e$ in the definition of `letcc` as in: `letcc` $k$ : Id $e$ : Expr : T. The type checker can then verify at each use site that the expression has the type T.

In the `letcc` example there is a further caveat, the identifier $k$ is provided to the macro as argument, and the macro user will expect to be able to use the identifier in the expression $e$ which is the intended behavior. As an example, suppose we apply `letcc`

to the arguments $f$ and $f(42)$ as in `letcc` $f\ f(42)$. The identifier will be declared before we execute the expression, which is what we want. However, the identifier will not be part of the type environment at the use site, because $f$ is not declared there. We need to collect this binding information in order to extend the type environment at the use sites and do the type check. We initially require the programmer to provide the binding annotations, but we expect to be able to devise an analysis in the future which can compute the needed binding information. `letcc` $k : \mathrm{Id}\ e : \mathrm{Expr} : \{k : \mathrm{Object} \to \mathrm{T}\} : \mathrm{T}$, where we use Object as the super type of all types. The grammar on the right of Figure 5.1 shows the fully type annotated `letcc` example.

The binding information problem is more complex than sketched here, because the expression may be used multiple times, in which case the control flow of the macro body determines which identifiers are declared. An analysis would need to compute a safe approximation on the binding information available in all contexts. If an identifier is defined in all context the analysis could compute a type for the identifier as the largest subtype of all the types it is declared or inferred to have, and issue an error if an identifier is not declared in all contexts.

```
grammar LetCC extends
        { Expression, Identifier }
  Expr |:=
    letcc k : Id e : Expr
      ⇒
    ◁[ label L
      fn k(v) ⇒ exit L with v
      e
      end L ]▷
  end
grammar LetCC extends
        { Expression, Identifier }
  Expr : T |:=
    letcc k : Id e : Expr{k : Object → T} : T
      ⇒
    ◁[ label L
      fn k(v) ⇒ exit L with v
      e
      end L ]▷
end
```

Figure 5.1: A grammar definition of a syntax for `letcc`, with and without type annotations on the left and right, respectively

Another contrived, but interesting example is shown in Figure 5.2, where we define the `mania` macro which takes a possible empty sequence of expressions as argument. If the sequence is empty it returns the declaration of object O with a method *foo* which takes an argument of type Boolean and if the sequence is non-empty it returns the declaration of object O but with a definition of the method *foo* where the argument type is $\mathbb{Z}32$. Type checking a program which makes use of the `mania` macro is difficult because in some contexts the argument *foo* has one type and in other contexts it has another. Figure 5.3 shows an example, where we apply the `mania` macro to

the expression 42, and thus get O defined with a *foo* function that takes a $\mathbb{Z}32$ as argument. However on the next line of the program we instantiate O and call the *foo* function with the value `true` which is a Boolean. This is a type error (we assume Boolean and $\mathbb{Z}32$ have been suitably defined).

```
grammar Mania extends { Declaration }
 Decl |:=
   mania e:Expr* ⇒
   case e of
     Empty ⇒ ◁[ object O[[]]() foo(b : Boolean) = b end ▷
     Cons(e1, e2) ⇒ ◁[ object O[[]]() foo(s : ℤ32) = s end ▷
   end
end
```

Figure 5.2: A macro definition which define the object O with a method *foo* who's type depends on the number of expressions

```
mania 42
O().foo(true)
```

Figure 5.3: An unsound and illegal use of the `mania` macro

The example shows that macros can influence the argument types of methods. Macros can also influence which types (objects and traits) are defined, the static arguments of objects and traits, the arguments to the object constructor, the extends relationship, and the set and types of methods. Our solution to this problem is a static analysis of macros, which we will discuss briefly in Section 5.6.

The two examples above illustrate some interesting challenges when it comes to type checking macros. Our goal is to find useful solutions to these problems and create a sound type system. We do so by defining the expansion algorithm as a transformation function from a core calculus with macros called Basic Core Fortress with Macros (BCFM) to a core calculus without macros called Basic Core Fortress (BCF). BCF is already defined as part of the Fortress Language Specification [7]. Then we define a type system for BCFM and prove that the expansion preserves these types, by means of progress and preservation theorems.

Type checking is relevant for most syntactic forms, not only expressions but also trait, object, and function definitions, where all the methods and fields must be type checked. Our work does not ensure that macro definitions are syntactic well-formed like in [32], e.g. that a macro does not try to compose an invalid abstract syntax tree (AST).

The Chapter is organized as follows: In Section 5.2 we restate the Basic Core Fortress (BCF) calculus for easy reference. Section 5.3 introduces the BCFM calculus. In Section 5.4 we present a formal description of the complete expansion algorithm which translates BCFM terms into BCF terms. In Section 5.5 we present a type system for BCFM and we state the theorems that needs to be proven to show that the expansion algorithm preserves types. We reflect on what we have learned so far and future work in Section 5.6, and conclude in Section 5.7.

## 5.2 Basic Core Fortress

In this Section we restate Basic Core Fortress (BCF) [7] for easy reference. We will be using BCF as the target language of the expansion algorithm described in Section 5.4, and BCF will be the foundation for BCFM described in Section 5.3. Figure 5.4 shows the abstract syntax of the calculus, which has been designed such that BCF programs are valid Fortress programs. A BCF program consists of zero or more *definitions* $\overline{d}$ followed by an *expression* $e$, called the *main expression*. We will be using the shorthand notation $\overline{x}$ throughout the chapter for a possibly comma separated sequence of $x$'s, it will be clear from the context when the commas are present. We write $x_i$ when referring to some element in the sequence $\overline{x}$, and if $\overline{x}$ and $\overline{y}$ are sequences then we write $\overline{x : y}$ for the sequence with elements $x_i : y_i$ made by applying the operator : to the elements of $\overline{x}$ and $\overline{y}$ pairwise. For brevity we also write $\lhd$ instead of the reserved word `extends`. A definition is either a *trait* or an *object definition*. A trait definition *td* consists of the reserved word `trait` followed by the name of the trait, *static parameters* $[\![\ \overline{\alpha \lhd \mathrm{N}}\ ]\!]$, an extends clause $\lhd \{\ \overline{\mathrm{N}}\ \}$, a number of *method definitions fd*, and the reserved word `end`. The static parameters is a sequence of *type variables* $\alpha$ and their *upper type bound* N. The upper bound can be the name of a trait $\mathrm{T}[\![\ \overline{\tau}\ ]\!]$ instantiated with *types* $\overline{\tau}$ or the reserved word `Object` which is the topmost type of the type hierarchy. An extends clause consists of a comma separated list of *type names* N which the trait extends. A method definition *fd* consists of the name of the method, static parameters, a comma separated sequence of *arguments* ( $\overline{x : \tau}$ ), a *return type* $\tau$, and the *method body* $e$. The arguments consist of a variable $x$ and a type $\tau$.

An object definition *od* is similar to a trait definition except for the reserved word `object`, and *constructor arguments* ( $\overline{x : \tau}$ ).

There are five kinds of expressions $e$; *variable* $x$, *self reference* `self`, *object instantiation* $\mathrm{O}[\![\ \overline{\tau}\ ]\!](\ \overline{e}\ )$, *field access* $e.x$, and *method invocation* $e.f[\![\ \overline{\tau}\ ]\!](\ \overline{e}\ )$.

The dynamic semantics of BCF programs is not particularly relevant in this work, and we refer the reader to the Fortress Language Specification [7] for the details. The static semantics will be presented briefly in Section 5.5.

| | | | |
|---|---|---|---|
| $\alpha$, $\beta$ | | | type variables |
| $f$ | | | method name |
| $x$ | | | field name |
| T | | | Trait name |
| O | | | Object name |
| $\tau$, $\tau'$, $\tau''$ | ::= | $\alpha$ | type |
| | \| | $\sigma$ | |
| $\sigma$ | ::= | $N$ | |
| | \| | $\mathrm{O}[\![\ \overline{\tau}\ ]\!]$ | |
| $N$, $K$, $L$ | ::= | $\mathrm{T}[\![\ \overline{\tau}\ ]\!]$ | |
| | \| | Object | |
| $e$ | ::= | $x$ | expressions |
| | \| | `self` | |
| | \| | $\mathrm{O}[\![\ \overline{\tau}\ ]\!](\ \overline{e}\ )$ | |
| | \| | $e.x$ | |
| | \| | $e.f[\![\ \overline{\tau}\ ]\!](\ \overline{e}\ )$ | |
| $fd$ | ::= | $f[\![\ \overline{\alpha \lhd \mathrm{N}}\ ]\!]\ (\ \overline{x:\tau}\ ){:}\tau = e$ | |
| $td$ | ::= | `trait` $\mathrm{T}[\![\ \overline{\alpha \lhd \mathrm{N}}\ ]\!] \lhd \{\ \overline{\mathrm{N}}\ \}\ \overline{fd}$ `end` | |
| $od$ | ::= | `object` $\mathrm{O}[\![\ \overline{\alpha \lhd \mathrm{N}}\ ]\!](\ \overline{x:\tau}\ ) \lhd \{\ \overline{\mathrm{N}}\ \}\ \overline{fd}$ `end` | |
| $d$ | ::= | $td$ | definitions |
| | \| | $od$ | |
| $p$ | ::= | $\overline{d}\ e$ | program |

Figure 5.4: Abstract syntax for Basic Core Fortress

## 5.3 Basic Core Fortress with Macros

In this Section we formally introduce the *Basic Core Fortress with Macros* (BCFM) calculus. BCFM is a superset of the BCF calculus and plays the same role in the syntax normalization process as node expressions described in Chapter 4. A node expression is the result of the parsing stage and will undergo expansion in the transformation stage. Similarly, BCFM programs will be the result of the parsing stage, and subject to expansion in the transformation stage.

The abstract syntax of BCFM is shown in Figure 5.5. A BCFM program consists of zero or more *macro definitions* $\overline{d}$, zero or more *macro terms* $\overline{mt}$, and then a macro term $mt$, called the *main term*. The structure is inspired by BCF and the macro terms $\overline{mt}$ are intended to expand to BCF *definitions* $\overline{d}$ and the main term is intended to expand to a BCF expression, however we don't give any guaranties that they will be well-behaved and expand as intended. We expect a BCFM program to be constructed by a process very similar to the parsing stage in Chapter 4 and so the programs are well-behaved by construction.

A macro definition is similar to a *Transformer* in Chapter 4 which is a named macro with arguments (a sequence of pattern variables). A macro definition consists of the keyword `macro`, a name $M$, static arguments $[\![\ \overline{\alpha \lhd \mathrm{N}}\ ]\!]$, a comma separated sequence of *pattern variable typings* $(\ \overline{pvt}\ )$, a *return type* $\tau$, and the *macro body* $mt$. We distinguish between expressions and other syntactic forms by defining two different kinds of pattern variable typings, one for expressions and one for the other syntactic forms, thus a pattern variable typing $pvt$ is either a pattern variable *expression* typing

$pv : \eta : \tau$ or a pattern variable *non-expression* typing $pv$. An expression typing consists of a pattern variable $pv$, a set of bindings $\eta$, and a type $\tau$. A binding $x : \tau$ is a variable $x$ and its type $\tau$. A non-expression typing consists of just a pattern variable $pv$.

Macro terms are very similar to node expressions in Figure 4.8 extended with explicit constructors for *node constructors* $nc$, empty lists, non-empty lists, and definitions $d$. Macro invocation and ellipsis are two exceptions where macro invocations also have static arguments and the second argument of the *ellipsis constructor* Ellipsis is a list where $\langle\, mt \,\rangle = \text{Cons}(\ mt, \text{Empty}\ )$, $\langle\,\rangle = \text{Empty}$, and $\langle mt_1, mt_2, \ldots, mt_n \rangle = \text{Cons}(\ mt_1, \text{Cons}(\ mt_2, \ldots, \text{Cons}(\ mt_n, \text{Empty}\ ) \ldots\ )\ )$.

Node constructors $nc$ and definitions are similar to expressions and definitions in BCF, except they refer to macro terms where the BCF counterparts would refer to expressions. One could imagine writing a macro which would produce the static arguments of a method definition, but we have not found any use for it yet and we want to focus on the core properties of type checking in our macro system.

| | | | |
|---|---|---|---|
| $\alpha$, $\beta$ | | | type variables |
| $f$ | | | method name |
| $x$ | | | field name |
| $pv$ | | | pattern variable |
| T | | | Trait name |
| O | | | Object name |
| C | | | Trait or object name |
| $M$ | | | Macro name |
| $\tau$, $\tau'$, $\tau''$ | ::= | $\alpha$ | type |
| | \| | $\sigma$ | |
| $\sigma$ | ::= | $N$ | |
| | \| | $\mathrm{O}[\![ \overline{\tau} ]\!]$ | |
| $N$, $K$, $L$ | ::= | $\mathrm{T}[\![ \overline{\tau} ]\!]$ | |
| | \| | Object | |
| $fd$ | ::= | $f[\![ \overline{\alpha \lhd \mathrm{N}} ]\!] ( \overline{x : \tau} ) : \tau = mt$ | |
| $td$ | ::= | `trait` $\mathrm{T}[\![ \overline{\alpha \lhd \mathrm{N}} ]\!] \lhd \{ \overline{\mathrm{N}} \}$ $\overline{fd}$ `end` | |
| $od$ | ::= | `object` $\mathrm{O}[\![ \overline{\alpha \lhd \mathrm{N}} ]\!] ( \overline{x : \tau} ) \lhd \{ \overline{\mathrm{N}} \}$ $\overline{fd}$ `end` | |
| $d$ | ::= | $td$ | definitions |
| | \| | $od$ | |
| $nc$ | ::= | $x$ | node constructors |
| | \| | *self* | |
| | \| | $\mathrm{O}[\![ \overline{\tau} ]\!] ( \overline{mt} )$ | |
| | \| | $mt.x$ | |
| | \| | $mt.f[\![ \overline{\tau} ]\!] ( \overline{mt} )$ | |
| $mt$ | ::= | $pv$ | Macro terms |
| | \| | Empty | empty list |
| | \| | Cons( $mt$, $mt$ ) | list constructor |
| | \| | `case` $mt$ `of`<br>Empty $\Rightarrow$ $mt$<br>Cons( $pv$, $pv$ ) $\Rightarrow$ $mt$<br>`end` | case dispatch |
| | \| | Ellipsis( $mt, \langle \overline{mt} \rangle$ ) | ellipsis |
| | \| | $M[\![ \overline{\tau} ]\!] ( \overline{mt} )$ | macro invocation |
| | \| | $d$ | |
| | \| | $nc$ | |
| $pvt$ | ::= | $pv : \eta : \tau$ | |
| | \| | $pv$ | |
| $md$ | ::= | `macro` $M[\![ \overline{\alpha \lhd \mathrm{N}} ]\!] ( \overline{pvt} ) : \tau \Rightarrow mt$ | macro definition |
| $p$ | ::= | $\overline{md}$ $\overline{mt}$ $mt$ | program |

Figure 5.5: Abstract syntax for Basic Core Fortress with Macros

## 5.4 Expansion Algorithm

In this Section we present the complete expansion algorithm which rewrites BCFM programs into BCF programs. In comparison to the algorithm presented in Chapter 4 we also include the rules for node constructors besides the rules for the new constructs in BCFM.

The expansion algorithm is a transformation from BCFM programs to BCF programs. The algorithm is specified as a small-step operational semantics [74] which is defined by two transition systems. The first system describes the translation of BCFM programs $p$ and the second system describes the translation of macro terms.

Both transition systems consist of a set of configurations $\sigma \in \Sigma$, a set of terminal configurations $\mathrm{T} \subseteq \Sigma$, and a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$.

Configurations in the first system are simply BCFM programs which subsumes BCF programs, and BCF programs are terminal configurations. Configurations in the second system consists of two environments and a macro term. The environments are the *definition environment* $\Upsilon$ and the *pattern variable environment* $\Pi$. The definition environment $\Upsilon$ contain macro definitions, and the pattern variable environment $\Pi$ maps pattern variables to values. A value is a BCF expression or definition. A terminal configuration in the second system is a configuration where the macro term is a value. We write "$\Pi\,[pv \mapsto v]$" to denote an extension of $\Pi$ with the new binding "$pv \mapsto v$".

We start by describing the system for transforming programs, which consists of two rules [E-Program Terms] and [E-Program Term]:

[E-Program Terms]

$$\frac{\overline{md}; \emptyset \vdash mt_i \rightarrow \overline{md}; \emptyset \vdash mt_i'}{\vdash \overline{md}\ \overline{mt}\ mt \rightarrow \vdash \overline{md}\ \overline{mt'}\ mt}$$

[E-Program Term]

$$\frac{\overline{md}; \emptyset \vdash mt \rightarrow \overline{md}; \emptyset \vdash mt'}{\vdash \overline{md}\ \overline{d}\ mt \rightarrow \vdash \overline{md}\ \overline{d}\ mt'}$$

The rule [E-Program Terms] performs one step of the transformation of each of the macro terms using a definition environment consisting of all the macro definitions and an empty pattern variable environment. The order in which the terms are reduced does not matter, we only require that the order is deterministic. The reduction order is also irrelevant in other rules, and we use the same notation there and only assume that the used evaluation order is deterministic. Similarly rule [E-Program Term] performs one step of the transformation of the main term using the same environments.

Macro terms are transformed by using the second transition system, which is described in a number of rules for each syntactic construct. The rule [E-Pattern Variable] is evaluated by simply looking up the pattern variable in the pattern variable environment. The rules [E-Empty], [E-Cons], and [E-Case] are as one would expect, the empty list is the empty list, for the non-empty list we evaluate the first argument first $mt_1$ to a value, and then the second argument, for the case dispatch we first evaluate the argument [E-Case] to a value and if it is the empty list we evaluate $mt_1$ (rule [E-Case Empty]) and if it is not the empty list we evaluate $mt_2$ in a pattern variable environment where $pv_1$ is bound to $v_1$ and $pv_2$ is bound to $v_2$ (rule [E-Case Cons]).

[E-Pattern Variable]

$$\frac{\Pi(pv) = v}{\Upsilon; \Pi \vdash pv \rightarrow \Upsilon; \Pi \vdash v}$$

[E-Empty]

$$\Upsilon; \Pi \vdash \text{Empty}$$

[E-Cons]

$$\frac{\Upsilon; \Pi \vdash mt_1 -> \Upsilon; \Pi \vdash mt_1'}{\Upsilon; \Pi \vdash \text{Cons}(\ mt_1,\ mt_2\ ) \rightarrow \Upsilon; \Pi \vdash \text{Cons}(\ mt_1',\ mt_2\ )}$$

$$\frac{\Upsilon; \Pi \vdash mt_2 \rightarrow \Upsilon; \Pi \vdash mt_2'}{\Upsilon; \Pi \vdash \text{Cons}(\ v,\ mt_2\ ) \rightarrow \Upsilon; \Pi \vdash \text{Cons}(\ v,\ mt_2'\ )}$$

[E-Case]

$$\frac{\Upsilon; \Pi \vdash mt \rightarrow \Upsilon; \Pi \vdash mt'}{\begin{array}{l}\Upsilon; \Pi \vdash \texttt{case}\ mt\ \texttt{of} \\ \quad \text{Empty} \Rightarrow\ mt_1 \\ \quad \text{Cons}(\ pv_1,\ pv_2\ ) \Rightarrow\ mt_2 \\ \quad \texttt{end}\end{array} \rightarrow \begin{array}{l}\Upsilon; \Pi \vdash \texttt{case}\ mt'\ \texttt{of} \\ \quad \text{Empty} \Rightarrow\ mt_1 \\ \quad \text{Cons}(\ pv_1,\ pv_2\ ) \Rightarrow\ mt_2 \\ \quad \texttt{end}\end{array}}$$

[E-Case Empty]

$$\begin{array}{l}\Upsilon; \Pi \vdash \texttt{case}\ \text{Empty}\ \texttt{of} \\ \quad \text{Empty} \Rightarrow\ mt_1 \\ \quad \text{Cons}(\ pv_1,\ pv_2\ ) \Rightarrow\ mt_2 \\ \quad \texttt{end}\end{array} \rightarrow \Upsilon; \Pi \vdash mt_1$$

[E-Case Cons]

$$\begin{array}{l}\Upsilon; \Pi \vdash \texttt{case}\ \text{Cons}(\ v_1,\ v_2\ )\ \texttt{of} \\ \quad \text{Empty} \Rightarrow\ mt_1 \\ \quad \text{Cons}(\ pv_1,\ pv_2\ ) \Rightarrow\ mt_2 \\ \quad \texttt{end}\end{array} \rightarrow \Upsilon; \Pi\,[pv_1 \mapsto v_1]\,[pv_2 \mapsto v_2] \vdash mt_2$$

The rules for macro invocation [E-Macro Invocation] are almost the same as in Chapter 4 except that we use the function pvs to compute the sequence of pattern variables from the sequence of pattern variable typings, we do so by a simple case match on the two kinds of typing.

[E-Macro Invocation]

$$\frac{\Upsilon; \Pi \vdash mt_i \rightarrow \Upsilon; \Pi \vdash mt_i'}{\Upsilon; \Pi \vdash M(\overline{mt}) \rightarrow \Upsilon; \Pi \vdash M(\overline{mt'})}$$

$$\frac{\texttt{macro}\ M[\![\ \overline{\alpha <: N}\ ]\!](\ \overline{pvt}\ ) : \tau_0 \Rightarrow mt \in \Upsilon \qquad \overline{pv} = \text{pvs}(\ \overline{pvt}\ )}{\Upsilon; \Pi \vdash M(\overline{v}) \rightarrow \Upsilon; \Pi[\overline{pv \mapsto v}] \vdash mt}$$

The rules for ellipsis have not changed besides minor notational differences.

[E-Ellipses First]

$$\frac{|\overline{v}| + 1 = i \leq size_\Pi(\ mt\ ) \qquad \Pi' = extend_\Pi(\ PV(mt),\ i\ ) \qquad \Upsilon, \Pi' \vdash mt \rightarrow \Upsilon, \Pi' \vdash mt'}{\Upsilon; \Pi \vdash \text{Ellipses}(mt, \langle \overline{v} \rangle) \rightarrow \Upsilon; \Pi \vdash \text{Ellipses}(mt, \langle \overline{v} mt' \rangle)}$$

[E-Ellipses Second]

$$\frac{|\overline{v}| = size_{\Pi}(\ mt\ )}{\Upsilon, \Gamma \vdash \text{Ellipses}(mt, \langle \overline{v} \rangle) \rightarrow \Upsilon, \Gamma \vdash \overline{v}}$$

**where** $size_{\Gamma}(n) \ = max(|\Pi(pv_j)|) \quad \forall pv_j \in PV(n)$

$$extend_{\Pi}(\{\overline{pv_j}\}, i) = \begin{cases} pv_j \mapsto \Pi(pv_j) & \text{if } |\Pi(pv_j)| = 1 \\ pv_j \mapsto \Pi(pv_j).nth(i) & \text{if } |\Pi(pv_j)| > 1 \end{cases}$$

We will not go into details about the rules for transforming node constructors and definitions because they are straight forward, evaluate the various subterms and thread the environments along.

[E-Variable]

$$\Upsilon; \Pi \vdash x \rightarrow \Upsilon; \Pi \vdash x$$

[E-Self]

$$\Upsilon; \Pi \vdash \texttt{self} \rightarrow \Upsilon; \Pi \vdash \texttt{self}$$

[E-Object]

$$\frac{\Upsilon; \Pi \vdash mt_i \rightarrow \Upsilon; \Pi \vdash mt'_i}{\Upsilon; \Pi \vdash \text{O}[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ ) \rightarrow \Upsilon; \Pi \vdash \text{O}[\![\ \overline{\tau}\ ]\!](\ \overline{mt'}\ )}$$

[E-Method]

$$\frac{\Upsilon; \Pi \vdash mt \rightarrow \Upsilon; \Pi \vdash mt'}{\Upsilon; \Pi \vdash mt.f[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ ) \rightarrow \Upsilon; \Pi \vdash mt'.f[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ )}$$

$$\frac{\Upsilon; \Pi \vdash mt_i \rightarrow \Upsilon; \Pi \vdash mt'_i}{\Upsilon; \Pi \vdash v.f[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ ) \rightarrow \Upsilon; \Pi \vdash v.f[\![\ \overline{\tau}\ ]\!](\ \overline{mt'}\ )}$$

[E-Trait Definition]

$$\frac{\Upsilon; \Pi \vdash fd_i \rightarrow \Upsilon; \Pi \vdash fd'_i}{\begin{array}{c}\Upsilon; \Pi \vdash \texttt{trait}\ \text{T}[\![\ \overline{\alpha \lhd \text{N}}\ ]\!] \lhd \{\ \overline{N}\ \}\ \overline{fd}\ \texttt{end} \\ \rightarrow \\ \Upsilon; \Pi \vdash \texttt{trait}\ \text{T}[\![\ \overline{\alpha \lhd \text{N}}\ ]\!] \lhd \{\ \overline{N}\ \}\ \overline{fd'}\ \texttt{end}\end{array}}$$

[E-Object Definition]

$$\frac{\Upsilon; \Pi \vdash fd_i \rightarrow \Upsilon; \Pi \vdash fd'_i}{\begin{array}{c}\Upsilon; \Pi \vdash \texttt{object}\ \text{O}[\![\ \overline{\alpha \lhd \text{N}}\ ]\!](\ \overline{x : \tau}\ ) \lhd \{\ \overline{N}\ \}\ \overline{fd}\ \texttt{end} \\ \rightarrow \\ \Upsilon; \Pi \vdash \texttt{object}\ \text{O}[\![\ \overline{\alpha \lhd \text{N}}\ ]\!](\ \overline{x : \tau}\ ) \lhd \{\ \overline{N}\ \}\ \overline{fd'}\ \texttt{end}\end{array}}$$

[E-Method Definition]

$$\frac{\Upsilon; \Pi \vdash mt \rightarrow \Upsilon; \Pi \vdash mt'}{\begin{array}{c}\Upsilon; \Pi \vdash f[\![\ \overline{\alpha \lhd \text{N}}\ ]\!] \lhd (\ \overline{x : \tau}\ ) : \tau = mt \\ \rightarrow \\ \Upsilon; \Pi \vdash f[\![\ \overline{\alpha \lhd \text{N}}\ ]\!] \lhd (\ \overline{x : \tau}\ ) : \tau = mt'\end{array}}$$

## 5.5 A Type System for BCFM

In this Section we give an overview of our proposal for a type system for BCFM. We build upon and extend the type system for BCF, where we add new rules for the new constructs introduced in BCFM. We will only give detailed description of the new rules, and we refer the reader to the Fortress Language Specification [7] for details on the type rules which are shared among BCFM and BCF.

We start by describing the various environments we use in the rules, then we describe the rules of the system, and we finish by stating progress and preservation theorems for the type system with respect to the expansion algorithm.

The type system makes use of four environments; the *definition environment* $\Upsilon$, the *type bounds* environment $\Delta$, the *type environment* $\Gamma$, and the *pattern variable type environment* $\Pi$. The definition environment contains a set of types (object and trait definitions) and macro definitions. We generally use the definition environment when we need to ensure the presence of a definition like in rule [T-Macro Invocation] and [T-Field]. The type bounds environment contains a sequence of type variables and their bounds $\alpha <: N$, e.g. see rule [T-Trait Definition]. The type environment maps Fortress variables to a type $x : \tau$. The pattern variable environment maps pattern variables to their type $pv : \tau$. We have separated the pattern variables and the Fortress variables into two distinct type environments to make the distinction between the two clear.

We check the type correctness of a BCFM program using the rule [T-Program] where we initialize the definition environment by means of a yet to be specified static analysis which will compute a conservative approximation of the types defined by the macros defined in the program. A program is type correct if the macro definitions and the macro terms are well-typed (Ok), and the main term has type $\tau$ using the computed definition environment and the empty type bounds, type, and pattern variable environments.

[T-Program]

$$\frac{\begin{array}{c}\Upsilon = \text{analyze}(\overline{md}, \overline{md}) \\ \Upsilon \vdash \overline{md}\ \texttt{Ok} \qquad \Upsilon;\emptyset;\emptyset;\emptyset \vdash \overline{mt}\ \texttt{Ok} \qquad \Upsilon;\emptyset;\emptyset;\emptyset \vdash mt : \tau\end{array}}{\overline{md}\ \overline{mt}\ mt : \texttt{Ok}}$$

A macro definition is well-typed if the [T-Macro Definition] rule applies, which is very similar to the rule for trait and object definitions ([T-Trait Definition] and [T-Object Definition]) which are the same as for BCF. First we define the type bounds environment and use it to check that the static arguments $\overline{N}$ are well-formed (for trait and object definition we also check the extends clause). We then compute a sequence of elements consisting of a pattern variable, its bindings, and its type from the pattern variable typing using the function pvTypes defined as follows:

$$\begin{array}{lcl}\text{pvTypes}(pv : \eta : \tau, \text{ls}) & = & [pv : \eta : \tau]\,\text{pvTypes}(\ \text{ls}\ ) \\ \text{pvTypes}(pv, \text{ls}) & = & \text{pvTypes}(\ \text{ls}\ )\end{array}$$

If it is an expression typing then we add the pattern variable, the bindings, and the type to the sequence computed by applying pvTypes recursively on the rest of the sequence of the pattern variable typing. If the typing is a non-expression typing then we just apply pvTypes recursively.

We check the well-formedness of the types computed by pvTypes and then compute the type $\tau'$ of the macro body under the assumption that the pattern variables have the type computed by pvTypes, and we check that the declared return type of the macro $\tau_0$ is well-formed and that $\tau'$ is a subtype of $\tau_0$.

[T-Macro Definition]

$$\frac{\begin{array}{c}\Delta = \overline{\alpha <: N} \qquad \Upsilon;\Delta \vdash \overline{N}\ \texttt{Ok} \qquad \overline{pv : \eta : \tau} = \text{pvTypes}(\ \overline{pvt}\ ) \\ \Upsilon;\Delta \vdash \overline{\tau}\ \texttt{Ok} \qquad \Upsilon;\Delta;\emptyset;[\overline{pv : \tau}] \vdash mt : \tau' \qquad \Upsilon;\Delta \vdash \tau_0\ \texttt{Ok} \qquad \Upsilon;\Delta \vdash \tau' <: \tau_0\end{array}}{\Upsilon \vdash \texttt{macro}\ M[\![\ \overline{\alpha \lhd N}\ ]\!](\ \overline{pvt}\ ) : \tau_0 \Rightarrow mt\ \texttt{Ok}}$$

[T-Trait Definition]

$$\frac{\begin{array}{c}\Delta = \overline{\alpha <: N} \qquad \Upsilon;\Delta \vdash \overline{N}\ \texttt{Ok} \\ \Upsilon;\Delta \vdash \overline{L}\ \texttt{Ok} \qquad \Upsilon;\Delta;[\texttt{self} : \mathrm{T}[\![\ \overline{\alpha}\ ]\!]\,];\Pi;\mathrm{T} \vdash \overline{fd}\ \texttt{Ok} \qquad \text{oneOwner}_\Upsilon(\ \mathrm{T}\ )\end{array}}{\Upsilon;\Delta;\Gamma;\Pi \vdash \texttt{trait}\ \mathrm{T}[\![\ \overline{\alpha \lhd N}\ ]\!] \lhd \{\ \overline{L}\ \}\ \overline{fd}\ \texttt{end}\ \texttt{Ok}}$$

[T-Object Definition]

$$\frac{\begin{array}{c}\Delta = \overline{\alpha <: N} \\ \Upsilon;\Delta \vdash \overline{N}\ \texttt{Ok} \qquad \Upsilon;\Delta \vdash \overline{\tau}\ \texttt{Ok} \qquad \Upsilon;\Delta \vdash \overline{L}\ \texttt{Ok} \qquad \Upsilon;\Delta;\Gamma\,[\overline{x : \tau}]\,;\Pi \vdash \overline{vd}\ \texttt{Ok} \\ \Upsilon;\Delta;\Gamma\,[\texttt{self} : \mathrm{O}[\![\ \overline{\alpha}\ ]\!]\,][\overline{x : \tau}]\,;\Pi;\mathrm{O} \vdash \overline{fd}\ \texttt{Ok} \qquad \text{oneOwner}_\Upsilon(\ \mathrm{T}\ )\end{array}}{\Upsilon;\Delta;\Gamma;\Pi \vdash \texttt{object}\ \mathrm{O}[\![\ \overline{\alpha \lhd N}\ ]\!](\overline{x : \tau}) \lhd \{\ \overline{L}\ \}\ \overline{vd}\ \overline{fd}\ \texttt{end}\ \texttt{Ok}}$$

The rules for [T-Method Definition]] and method overriding ([Override]]) are the same as for BCF.

[T-Method Definition]

$$\frac{\begin{array}{c}\_\,\mathrm{C}\,\_ \lhd \{\ \overline{L}\ \}\ \_ \in \Upsilon \qquad \Upsilon;\Delta \vdash \text{override}(f, \{\ \overline{L}\ \}, [\![\ \overline{\alpha \lhd N}\ ]\!]\overline{\tau} \to \tau_0) \\ \Delta' = \Delta\ [\overline{\alpha <: N}] \qquad \Upsilon;\Delta' \vdash \overline{N}\ \texttt{Ok} \qquad \Upsilon;\Delta' \vdash \overline{\tau}\ \texttt{Ok} \\ \Upsilon;\Delta' \vdash \tau_0\ \texttt{Ok} \qquad \Upsilon;\Delta';\Gamma\,[\overline{x : \tau}]\,;\Pi \vdash mt : \tau' \qquad \Upsilon;\Delta' \vdash \tau' <: \tau_0\end{array}}{\Upsilon;\Delta;\Gamma;\Pi;\mathrm{C} \vdash f[\![\ \overline{\alpha \lhd N}\ ]\!](\overline{x : \tau}) : \tau_0 = mt\ \texttt{Ok}}$$

Method overriding: $\boxed{\Upsilon;\Delta \vdash \text{override}(f, \{\ \overline{L}\ \}, [\![\ \overline{\alpha \lhd N}\ ]\!]\overline{\tau} \to \tau)}$

[Override]

$$\frac{\begin{array}{c}\bigcup_{L_i \in \{\overline{L}\}} \text{mtype}_\Upsilon(\ f,\ L_i\ ) = [\![\ \overline{\beta \lhd K}\ ]\!]\overline{\tau'} \to \tau_0' \\ \overline{N} = [\overline{\alpha/\beta}]\overline{K} \qquad \overline{\tau} = [\overline{\alpha/\beta}]\overline{\tau'} \qquad \Upsilon;\overline{\alpha <: N} \vdash \tau_0 <: [\overline{\alpha/\beta}]t_0'\end{array}}{\Upsilon;\Delta \vdash \text{override}(f, \{\ \overline{L}\ \}, [\![\ \overline{\alpha \lhd N}\ ]\!]\overline{\tau} \to \tau_0)}$$

We now present the five new rules we have added for macro term typing; [T-Pattern Variable], [T-Empty], [T-Cons], [T-Case], and [T-Macro Invocation]. The first four rules are standard rules as one would expect. If the term is a pattern variable we look it up in the pattern environment and get its type. If the term Empty then it has the type Empty. If the term is Cons then it has the type List$[\![\ \tau\ ]\!]$ if all the elements in the list has type $\tau$. We assume that the types Empty and List$[\![\ \tau\ ]\!]$ are build-in types s.t. Empty$[\![\ \tau\ ]\!] \lhd$ List$[\![\ \tau\ ]\!]$. The rule for case dispatch is also straight forward, first we check that the argument $mt_0$ has type List$[\![\ \tau\ ]\!]$, then we check the subterm $mt_1$ in the same environments, then the subterm $mt_2$ in environments where $pv_1$ and $pv_2$ are bound to $\tau'$ and List$[\![\ \tau'\ ]\!]$ respectively, and finally we check to see if the types of the subterms are subtypes of the resulting type of the case dispatch.

The interesting part of macro term typing is the rule for macro invocation [Macro Invocation]. The rule is very similar to the rule for method invocation because macro invocation is very similar to method invocation. We start by looking up the macro

definition in the definition environment $\Upsilon$. Then we check if the static arguments $\overline{\tau}$ are well-formed and that they are subtypes of the bounds on the static arguments where we substitute the static arguments for the type variables $\overline{\tau} <: [\overline{\tau/\alpha}]\overline{N}$. Then we compute the sequence of pattern variables, bindings, and types using pvTypes and use the bindings to compute the type of the arguments $\Upsilon; \Delta; \Gamma\ \overline{\eta}; \Pi \vdash \overline{mt : \tau''}$ and use the types to check that the computed types are indeed subtypes of the expected types $\overline{\tau''} <: [\overline{\tau/\alpha}]\overline{\tau'}$.

[T-Pattern Variable]

$$\Upsilon; \Delta; \Gamma; \Pi \vdash pv : \Pi(pv)$$

[T-Empty]

$$\Upsilon; \Delta; \Gamma; \Pi \vdash \text{Empty} : \text{Empty}[\![\ \tau\ ]\!]$$

[T-Cons]

$$\frac{\Upsilon; \Delta; \Gamma; \Pi \vdash mt_1 : \tau \qquad \Upsilon; \Delta; \Gamma; \Pi \vdash mt_2 : List[\![\ \tau\ ]\!]}{\Upsilon; \Delta; \Gamma; \Pi \vdash \text{Cons}(mt_1, mt_2) : \text{List}[\![\ \tau\ ]\!]}$$

[T-Case]

$$\frac{\begin{array}{c}\Upsilon; \Delta; \Gamma; \Pi \vdash mt_0 : \text{List}[\![\ \tau'\ ]\!] \qquad \Upsilon; \Delta; \Gamma; \Pi \vdash mt_1 : \tau_1 \\ \Upsilon; \Delta; \Gamma; \Pi\,[pv_1 : \tau']\,[pv_2 : \text{List}[\![\ \tau'\ ]\!]] \vdash mt_2 : \tau_2 \qquad \Upsilon; \Delta; \Gamma; \Pi \vdash \tau_i <: \tau\end{array}}{\begin{array}{ll}\Upsilon; \Delta; \Gamma; \Pi \vdash \texttt{case}\ mt_0\ \texttt{of} & : \tau \\ \quad \text{Empty} \Rightarrow mt_1 & \\ \quad \text{Cons}(pv_1, pv_2) \Rightarrow mt_2 & \\ \texttt{end} & \end{array}}$$

[T-Macro Invocation]

$$\frac{\begin{array}{c}\texttt{macro}\ M[\![\ \overline{\alpha <: N}\ ]\!](\ \overline{pvt}\ ) : \tau_0\ \_\ \texttt{end} \in \Upsilon \\ \Upsilon; \Delta \vdash \overline{\tau}\ \texttt{Ok} \qquad \Upsilon; \Delta \vdash \overline{\tau} <: [\overline{\tau/\alpha}]\overline{N} \\ \overline{pv : \eta : \tau'} = \text{pvTypes}(\ \overline{pvt}\ ) \qquad \Upsilon; \Delta; \Gamma\ \overline{\eta}; \Pi \vdash \overline{mt : \tau''} \qquad \Upsilon; \Delta \vdash \overline{\tau''} <: [\overline{\tau/\alpha}]\overline{\tau'}\end{array}}{\Upsilon; \Delta; \Gamma; \Pi \vdash M[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ ) : [\overline{\tau/\alpha}]\tau_0}$$

The rules for node constructors are the same as the rules for expressions in BCF, except for a minor naming difference. We write *mt* where the rules for BCF would have had an *e*.

[T-Var]

$$\Upsilon; \Delta; \Gamma; \Pi \vdash x : \Gamma(x)$$

[T-Self]

$$\Upsilon; \Delta; \Gamma; \Pi \vdash \texttt{self} : \Gamma(\texttt{self})$$

[T-Object]

$$\frac{\begin{array}{c}\texttt{object}\ \text{O}[\![\ \overline{\alpha \lhd \_}\ ]\!](\ \overline{\_ : \tau'}\ )\ \_\ \texttt{end} \in \Upsilon \\ \Upsilon; \Delta \vdash \text{O}[\![\ \overline{\tau}\ ]\!]\ \texttt{Ok} \qquad \Upsilon; \Delta; \Gamma; \Pi \vdash \overline{mt : \tau''} \qquad \Upsilon; \Delta \vdash \overline{\tau''} <: [\overline{\tau/\alpha}]\tau'\end{array}}{\Upsilon; \Delta; \Gamma; \Pi \vdash \text{O}[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ ) : \text{O}[\![\ \overline{\tau}\ ]\!]}$$

[T-Field]

$$\frac{\texttt{object O}[\![\ \overline{\alpha\ \triangleleft\ \_}\ ]\!](\ \overline{x:\tau}\ )\ \_\ \texttt{end} \in \Upsilon \qquad \Upsilon;\Delta;\Gamma;\Pi \vdash mt_0 : \tau_0 \qquad \mathrm{bound}_\Delta(\ \tau_0\ ) = \mathrm{O}[\![\ \overline{\tau'}\ ]\!]}{\Upsilon;\Delta;\Gamma;\Pi \vdash mt_0.x_i : [\overline{\tau'/\alpha}]\tau_i}$$

[T-Method]

$$\frac{\begin{array}{c}\Upsilon;\Delta;\Gamma;\Pi \vdash mt_0 : \tau_0 \\ \mathrm{mtype}_\Upsilon(\ f,\ \mathrm{bound}_\Delta(\ \tau_0\ )\ ) = \{\ [\![\ \overline{\alpha \triangleleft N}\ ]\!]\overline{\tau'} \to \tau_0'\ \} \qquad \Upsilon;\Delta \vdash \overline{\tau}\ \texttt{Ok} \\ \Upsilon;\Delta \vdash \overline{\tau} <: [\overline{\tau/\alpha}]\overline{N} \qquad \Upsilon;\Delta;\Gamma;\Pi \vdash \overline{mt : \tau''} \qquad \Upsilon;\Delta \vdash \overline{\tau''} <: [\overline{\tau/\alpha}]\overline{\tau'}\end{array}}{\Upsilon;\Delta;\Gamma;\Pi \vdash mt_0.f[\![\ \overline{\tau}\ ]\!](\ \overline{mt}\ ) : [\overline{\tau/\alpha}]\tau_0'}$$

The rules for method type lookup ([MT-Self], [MT-Super], and [MT-Obj]), sub typing ( [S-Refl], [S-Obj], [S-Trans], [S-Var], and [S-Tapp]), well-formed types ([W-Obj], [W-Var], and [W-Tapp]), bound of types, and check for one owner for all visible methods ([One Owner]) are all unchanged compared to BCF. The rules makes use of an auxiliary function for method name lookup:

$$\mathrm{Fname}(\ f[\![\ \overline{\alpha \triangleleft N}\ ]\!](\ \overline{x:\tau}\ ) : \tau = mt\ ) = f.$$

[MT-Self]

$$\frac{\_\ \mathrm{C}[\![\ \overline{\alpha \triangleleft \_}\ ]\!]\ \_\ \overline{fd}\ \_\ \in \Upsilon \qquad f[\![\ \overline{\beta \triangleleft N}\ ]\!](\_ : \tau') : \tau_0' = mt \in \{\ \overline{fd}\ \}}{\mathrm{mtype}_\Upsilon(\ f,\ \mathrm{C}[\![\ \overline{\tau}\ ]\!]\ ) = \{\ [\![\ \overline{\tau/\alpha}\ ]\!][\![\ \overline{\beta \triangleleft N}\ ]\!]\overline{\tau'} \to \tau_0'\ \}}$$

[MT-Super]

$$\frac{\_\ \mathrm{C}[\![\ \overline{\alpha \triangleleft \_}\ ]\!]\ \_\ \triangleleft \{\ \overline{N}\ \}\ \_\ \overline{fd}\ \_\ \in \Upsilon \qquad f \notin \{\ \overline{\mathrm{Fname}(\ fd\ )}\ \}}{\mathrm{mtype}_\Upsilon(\ f,\ \mathrm{C}[\![\ \overline{\tau}\ ]\!]\ ) = \bigcup_{N_i \in \{\ \overline{N}\ \}} \mathrm{mtype}_\Upsilon(\ f,\ [\![\ \overline{\tau/\alpha}\ ]\!]N_i\ )}$$

[MT-Obj]

$$\mathrm{mtype}_\Upsilon(\ f,\ \texttt{Object}\ ) = \emptyset$$

[S-Refl]

$$\Upsilon;\Delta \vdash \tau <: \tau$$

[S-Obj]

$$\Upsilon;\Delta \vdash \tau <: \texttt{Object}$$

[S-Trans]

$$\frac{\Upsilon;\Delta \vdash \tau_1 <: \tau_2 \qquad \Upsilon;\Delta \vdash \tau_2 <: \tau_3}{\Upsilon;\Delta \vdash \tau_1 <: \tau_3}$$

[S-Var]

$$\Upsilon;\Delta \vdash \alpha <: \Delta(\ \alpha\ )$$

[S-Tapp]

$$\frac{\_\ \mathrm{C}[\![\ \overline{\alpha \lhd \_}\ ]\!]\ \_ \lhd \{\ \overline{N}\ \}\ \_ \in \Upsilon}{\Upsilon; \Delta \vdash \mathrm{C}[\![\ \overline{\tau}\ ]\!] <: [\![\ \overline{\tau/\alpha}\ ]\!] N_i}$$

[W-Obj]

$$\Upsilon; \Delta \vdash \texttt{Object Ok}$$

[W-Var]

$$\frac{\alpha \in \mathrm{dom}(\ \Delta\ )}{\Upsilon; \Delta \vdash \alpha\ \texttt{Ok}}$$

[W-Tapp]

$$\frac{\_\ \mathrm{C}[\![\ \overline{\alpha \lhd N}\ ]\!]\ \_ \in \Upsilon \qquad \Upsilon; \Delta \vdash \overline{\tau}\ \texttt{Ok} \qquad \Upsilon; \Delta \vdash \overline{\tau} <: [\![\ \overline{\tau/\alpha}\ ]\!] \overline{N}}{\Upsilon; \Delta \vdash \mathrm{C}[\![\ \overline{\tau}\ ]\!]\ \texttt{Ok}}$$

Bound of type: $\boxed{\mathrm{bound}_\Delta(\ \tau\ ) = \sigma}$

$$\begin{array}{lcl} \mathrm{bound}_\Delta(\ \alpha\ ) & = & \Delta(\ \alpha\ ) \\ \mathrm{bound}_\Delta(\ \sigma\ ) & = & \sigma \end{array}$$

[One Owner]

$$\frac{\forall f \in \mathrm{visible}_\Upsilon(\ \mathrm{C}\ )\ .\ f \text{ only occurs once in } \mathrm{visible}_\Upsilon(\ \mathrm{C}\ )}{\mathrm{oneOwner}_\Upsilon(\ \mathrm{C}\ )}$$

$$\begin{array}{lcl} \mathrm{defined}_\Upsilon(\ \mathrm{C}\ ) & = & \{\ \overline{\mathrm{Fname}(\ Fd\ )}\ \} \\ & & \quad \text{where\_ C \_ } \overline{fd}\ \texttt{\_ end} \in \Upsilon \\ \mathrm{inherited}_\Upsilon(\ \mathrm{C}\ ) & = & \bigcup_{N_i \in \{\ \overline{N}\ \}} \{\ f_i \mid f_i \in \mathrm{visible}_\Upsilon(\ N_i\ ), f_i \notin \mathrm{defined}_\Upsilon(\ \mathrm{C}\ )\ \} \\ & & \quad \text{where } \_\ \mathrm{C}\ \_ \lhd \{\ \overline{N}\ \}\ \_ \in \Upsilon \\ \mathrm{visible}_\Upsilon(\ \mathrm{C}\ ) & = & \mathrm{defined}_\Upsilon(\ \mathrm{C}\ ) \cup \mathrm{inherited}_\Upsilon(\ \mathrm{C}\ ) \end{array}$$

We want to be able to do a modular type check of BCFM programs which guarantees that the resulting BCF programs will also be well-typed. We have already presented a modular type system for BCFM, and we intend to prove that the transformation algorithm preserves the types by means of the following *progress* and *preservation* theorems. We assume that BCFM programs are well-behaved by construction – the sequence of macro terms $\overline{mt}$ expands to a syntactically valid sequence of definitions, and the main term expands to a syntactically valid expression.

**Theorem 5.1** (Progress) *If* $\Upsilon; \Delta; \Gamma; \Pi \vdash mt : \tau$*, for some* $\tau$*, is a closed macro term then either mt is a value (a BCF term) or* $\Upsilon; \Delta; \Gamma; \Pi \vdash mt \rightarrow \Upsilon; \Delta; \Gamma; \Pi \vdash mt'$

**Theorem 5.2** (Preservation) *If* $\Upsilon; \Delta; \Gamma; \Pi \vdash mt : \tau$ *and* $\Upsilon; \Delta; \Gamma; \Pi \vdash mt \rightarrow \Upsilon; \Delta; \Gamma; \Pi \vdash mt'$*, for some macro term* $mt'$*, then* $\Upsilon; \Delta; \Gamma; \Pi \vdash mt' : \tau$*.*

## 5.6 Reflections and Future Work

In this Section we present some thoughts on the type system, typed macros in general, and future work.

The type system is an extension of the type system for BCF where we treat macro definitions and invocations in much the same way as method definitions and invocations. The interesting feature of the type system is that it allows us to type check interesting macros like the `letcc` example from Section 5.1, and that we have identified the need for the definition environment $\Upsilon$ and binding information on expressions.

The definition environment is needed because the macro definitions in a given program may introduce different types depending on the syntax of the program. The types may differ on the name of the types, the static arguments, the sequence of arguments (for objects), the extends clauses, the set of methods, the types on method arguments, and return types on methods. The expansion algorithm includes a case dispatch which gives rise to path sensitivity issues. We might define a set of correlated macros that when used properly produces a well-formed and well-typed program, the challenge here will be the same as for other analysis where path sensitivity is an issue.

A conservative approximation to the definition environment $\Upsilon$ can be computed in at least three different ways:

1. Each time there are two different definitions, then pick the most specific of the two
2. Compute a new conservative least-upper-bound defnition
3. Issue an error if we discover two possibly different definitions

We go for 2. since it is more precise than 1. which also involves defining what "most specific" means. We intend to compute a conservative approximation to the definition environment using the monotone framework, that will involve defining a partial order for all the parts (name, static arguments, arguments, extends clause, set of methods, and types on the methods) of a definition which may differ. The specification of the partial order will make use of the partial order introduced by the subtype relation and the extends clause. We leave the definition of the partial order as future work.

In the current proposal for the type system we require the macro-definer to annotate expression arguments with bindings for any free identifiers in the expression. However, the result of the type check may be unsound if the annotations are not correct. It would therefore be very useful if we could infer correct annotations to the benefit of the programmer. It is straight forward to compute the set of free identifiers in an expression, however their type may depend on the control flow and the contexts in which the expression occurs. However we believe that it is possible to compute a type for each such identifier as a greatest lower bound on the possible types. We leave the specifics of such an algorithm as future work.

The type system supports *polymorphic* or *generic* macros by means of type variables which can be used to give the arguments or the return value some polymorphic type. However, to what degree are polymorphic macros needed. As we saw in the `letcc` example we need parametric polymorphism when specifying what we call *control flow macros*. A control flow macro is a macro that introduces some new control flow in a program, like `letcc`. As another example lets look at the church encoding of booleans. We define true (`tru`) and false (`fls`) as two functions which take two arguments each and returns the first and the second argument, respectively. We define the type of the arguments to be integers ($\mathbb{Z}32$).

$$\texttt{tru} = \texttt{fn}\ \ t : \mathbb{Z}32 \Rightarrow \texttt{fn}\ \ f : \mathbb{Z}32 \Rightarrow t$$
$$\texttt{fls} = \texttt{fn}\ \ t : \mathbb{Z}32 \Rightarrow \texttt{fn}\ \ f : \mathbb{Z}32 \Rightarrow f$$

We first define the notational shorthand IntBool as a type for functions like `tru` and `fls` which takes two integers as arguments and return an integer. We then define the *cond* macro which takes three arguments, a function of type IntBool and two integers. The *cond* macro will apply the first argument to the two other arguments:

$$\text{IntBool} = \mathbb{Z}32 \to \mathbb{Z}32 \to \mathbb{Z}32$$
$$\texttt{macro}\ cond[\![](c : \{\} : \text{IntBool},\ t : \{\} : \mathbb{Z}32,\ f : \{\} : \mathbb{Z}32) : \mathbb{Z}32 \Rightarrow c(t(f))$$

The definition enables us to conditionally choose between two integer expressions: $cond(\texttt{tru}, 42, 84)$, that will evaluate to $42$ as expected. However, if we apply *cond* to three booleans: $cond(\texttt{tru}, \texttt{fls}, \texttt{tru})$ then it should and would be a type error because *cond* is not defined for booleans. We could easily make a new definition for *cond* which would allow us to conditionally choose between two booleans, but that would lead to code duplication.

The better solution would be to use polymorphism:

$$\texttt{macro}\ cond[\![\ T\ ]\!](c : \{\} : \mathrm{T} \to \mathrm{T} \to \mathrm{T}, t : \{\} : \mathrm{T}, f : \{\} : \mathrm{T}) : \mathrm{T}\ \Rightarrow\ c(t(f))$$

Which is a single definition and allows both uses of the macro: $cond(tru, 42, 84)$ and $cond(tru, fls, tru)$. Thus our conclusion, that control flow macros need parametric polymorphism.

## 5.7 Conclusion

We have presented the Basic Core Fortress (BCF) and Basic Core Fortress with Macros (BCFM) calculi along with a type system for BCFM, which prevents type errors in expanded macros. We also presented a complete description of the expansion algorithm from BCFM to BCF and formulated progress and preservation theorems with respect to the type system and the expansion algorithm. We identified the needs for a conservative approximation of which types are defined in a program with macros and the need for computing binding information associated with expression arguments to macro invocations. We leave the definition of how to compute these as future work along with the proofs for progress and preservation.

# Conclusion

*An investment in knowledge always pays the best interest*
— Benjamin Franklin

We conclude that it is feasible and useful to create programming languages with strong security guarantees for secure multiparty computation.

We have designed and implemented a high-level, domain-specific language called the Secure Multiparty Computation Language (SMCL), which allows programmers to express key concepts from the domain of secure multiparty computation precisely and without expert knowledge on how to design and implement cryptographic protocols.

We have developed a concurrent semantics for the entire SMCL language. The semantics is carefully crafted to ensure security of SMCL programs, along with an effect-based type system for SMCL which prevents SMCL programs from getting "stuck" and approximate our concept of hoistability. An SMCL program can get "stuck" as the result of an insecure operation. The type system also provides better precision than previous type systems for noninterference, and we prove its soundness.

SMCL programs are always secure against a broad spectrum of attacks, which exploits physical measurements an adversary may perform during the computation of an SMCL program. We defined the concept of *trace security* which is based on a formalization of physical measurements and we show that SMCL programs are trace-secure under the reasonable assumption that all the operations of the runtime we use are trace-secure.

We have also identified the concept of semantic security, where information may be revealed through values that are intentionally opened. It is the responsibility of the programmer of SMCL programs to ensure that such information is not a security problem. The programmer gets some help from the SMCL compiler in terms of warnings about the relationship among opened variables, however we issue no guarantees.

The lack of guarantees for semantic security lead us to work on the domain-specific language PySMCL which is embedded in Python. In PySMCL we require explicit definition of the intended ideal functionality in the program using two different kinds of `open` expressions, and we intend to verify that the program does indeed UC-realize the ideal functionality by automatically generating a machine checkable proof. We provide a proof stub for a lemma whenever information may leak in ways that are not explicitly allowed by the ideal functionality. The proof stub must be completed by the programmer in order to make the whole proof go through

We have described how we intend to embed PySMCL in Python in a seamless way by using function decorators and a careful choice of domain-specific value constructors. The sketched implementation of PySMCL will be faithful to the dynamic nature of Python by not requiring a separate compilation/verification step as verification and preprocessing is done dynamically as the file is loaded by the Python interpreter

We also provided a core calculus for PySMCL which we expect will be a useful stepping stone for defining formal versions of the suggested analysis along with proofs of the security properties of PySMCL programs.

All of this shows that it is feasible to create programming languages with strong security guarantees for secure multiparty computation. We have also shown that such languages can be useful. We have shown example SMCL programs which among others include generalizations of the seminal Millionaire's problem and a fully fledged implementation of a double auction which has been used in the very first large-scale practical application of secure multiparty computation. This is a huge testimony to the usefulness of SMCL. It is noteworthy that about 80% of the respondents in an on-line survey carried out in connection to the double auction said that it was important to them that the bids were kept confidential, and also that they were happy about the confidentiality that the system offered. Which means that confidentiality is seen as important by ordinary people, at least when money and clear conflict of interests is involved.

We have also worked on improving the usefulness of programs for secure multiparty computation by creating an optimization which potentially reduces the number of multiplications on secret values in an SMCL program and proved that it is correct. A multiplication on secret values may be up to 1000 times as slow as multiplications on non-secret values.

We also conclude that syntactic abstraction can enable programming languages to evolve over time by providing new syntax and semantics as they become needed.

We have designed a syntactic abstraction system for the Fortress programming language, which allows new syntax and semantics to be added in libraries as they become needed. The system enables the language to gracefully adapt to unanticipated needs as they become apparent over time.

Our system for syntax abstraction is based on parsing expression grammars, allows user-defined syntax to be indistinguishable from core Fortress syntax, and provides composition of independent extensions. The Fortress syntactic abstraction mechanism is so flexible that various language constructs can be moved from the core language syntax into libraries.

We have also designed an extension of the Basic Core Fortress (BCF) calculus called Basic Core Fortress with Macros (BCFM), and a type system for BCFM, which extend the static guarantees provided by the type system for BCF to cover BCFM. The type system ensures that use site type errors are caught at the invocation of macros and that no type errors occur in the expanded code if the macro definitions in a program are well typed.

We have stated progress and preservation theorems with respect to the type system and the expansion algorithm, and identified the needs for a conservative approximation of which types are defined in a BCFM program and the need for computing binding information associated with expression arguments to macro invocations. We leave the definition of how to compute these as future work along with the proofs for progress and preservation.

# Appendix A

## SMCL Goes Live

### A.1 Implementation of Double Auction

```
class abstract Farmer {
  String name;
}
class SellingFarmer
    extends Farmer {
  SellBid[ ] bids;
}
class SellBid {
  sint amount;
  sint contractId;
}
class BuyingFarmer
    extends Farmer {
  sint[ ] bids;
}
```

```
class AuctionResult {
  int mcp;
  int p1;
  int p2;
  int p3;
  int z1;
  int z2;
  int z3;
  int k;
  Contract[ ] contracts;
}
class Contract {
  int id;
  int amount;
}
```

Figure A.1: Classes used in the Double Auction example

```
declare server DoubleAuction: {
  tunnel of BuyingFarmer buyingFarmers;
  tunnel of SellingFarmer sellingFarmers;
```

Figure A.2: Double Auction implemented in SMCL

```
function sint[] computeAgrDemand() {
  sint[] agrDemand = new sint[4000];
  foreach (BuyingFarmer f in buyingFarmers) {
    if (checkBuyingFarmer(f)) {
      for (int inx=0; inx < agrDemand.length; inx++) {
        agrDemand[inx] += f.getBids(inx);
      }
    }
  }
}
```

Figure A.3: Implementation of aggregated demand

```
function sint[] computeAgrSupply() {
  sint[] agrSupply = new sint[4000];
  foreach (SellingFarmer f in sellingFarmers) {
    if (checkSellingFarmer(f)) {
      for (int inx=0; inx < agrSupply.length; inx++) {
        agrSupply[inx] += f.getBids(inx).getPrice();
      }
    }
  }
}
```

Figure A.4: Implementation of aggregated supply

```
function void main: {
  sint[ ] agrDemand = computeAgrDemand();
  sint[ ] agrSupply = computeAgrSupply();
  AuctionResult res = searchPrice(agrDemand, agrSupply);
  if (res.getP1() == agrDemand.length) {
    display(``P1 = 4000, so no computation of''
      + ``marginal bids is possible'';
    res.setMcp(getInput());
    openFinalBids(res);
    return;
  }
  // compute marginal bids
  int k = 0;
  int base = res.getP1();
  boolean repeat = promptComputeMarginal(res);
  while (repeat) {
    base += k;
    k = searchMarginal(base, agrDemand);
    boolean marg = openMarginalBids(k, base, res, agrDemand);
    if (!marg) {
      display(``No more marginal bids found.'');
      break;
    }
    // only continue if we end up at price less than 4000
    if (base < agrDemand.length - 1) {
      repeat = promptComputeMarginal(res);
      }
    else {
      break;
    }
  }
  res.setMcp(getInput());
  openFinalBids(res);
  return;
}
```

Figure A.5: Main function of the Double Auction server

```
function AuctionResult searchPrice
  (sint[] agrDemand, sint[] agrSupply) {
   AuctionResult res = new AuctionResult();
   int s1 = search1(agrDemand, agrSupply);
   int s2 = search2(agrDemand, agrSupply);
   if (s1 == s2) {
     int tz = open(agrDemand[s1] - agrSupply[s1] |
       agrDemand, agrSupply, s1);
     if (tz == 0) {
       res.setP1(s1);
       res.setP2(s1);
       res.setZ1(tz);
       res.setZ2(tz);
     } else if (tz > 0) {
       res.setP1(s1); // p2, z2 undefined
       res.setZ1(tz);
     } else {
       res.setP2(s1); // p1, z1 undefined
       res.setZ2(tz);
     }
   } else {
     res.setP1(s1);
     res.setP2(s2);
     res.setZ1(open(agrDemand[s1] - agrSupply[s1] |
       agrDemand, agrSupply, s1));
     res.setZ2(open(agrDemand[s2] - agrSupply[s1] |
       agrDemand, agrSupply, s2));
   }
   if (res.getP1() != null && res.getP2() != null) {
     double p3 = (res.getP2() + res.getP1()) / 2.0;
     res.setP3(p3);
     sint z3 = agrDemand[res.getP2()] - agrSupply[res.getP1()];
     res.setZ3(open(z3 | z3));
   }
   // open up and store agrDemand[p1], agrSupply[p1],
   // agrDemand[p2], agrSupply[p2], agrDemand[p3], agrSupply[p3]
   if (res.getP1() != null) {
     res.setDemand1(open(agrDemand[res.getP1()] | agrDemand, res));
     res.setSupply1(open(agrSupply[res.getP1()] | agrSupply, res));
   }
   if (res.getP2() != null) {
     res.setDemand2(open(agrDemand[res.getP2()] | agrDemand, res));
     res.setSupply2(open(agrSupply[res.getP2()] | agrSupply, res));
   }
   if (res.getP3() != null) {
    res.setDemand3(open(agrDemand[res.getP2()] | agrDemand, res));
    res.setSupply3(open(agrSupply[res.getP1()] | agrSupply, res));
   }
   return res;
}
```

Figure A.6: The searchPrice function computes the indicies of the potential

```
function int search1(sint[] agrDemand, sint[] agrSupply) {
  int low = 1;
  int high = agrSupply.length + 1;
  while (low + 1 < high) {
    int mid = (low + high + 1) / 2;
    sbool res = agrDemand[mid - 1] == (agrSupply[mid - 1]);
    if (open(res | res)) {
      low = mid;
    } else {
      high = mid;
    }
  }
  return low - 1;
}
```

Figure A.7: Binary search for the upper bound on the index of the market clearing price

```
function int search2(sint[] agrDemand, sint[] agrSupply) {
  int low = 0;
  int high = agrSupply.length;
  while (low + 1 < high) {
    int mid = (low + high) / 2;
    sint t = agrSupply[mid - 1] + 1;
    sbool res = agrDemand[mid - 1] == t;
    if (open(res | res)) {
      low = mid;
    } else {
      high = mid;
    }
  }
  return high - 1;
}
```

Figure A.8: Binary search for the lower bound on the index of the market clearing price

```
function int searchMarginal(int p, sint[] agrDemand) {
  int low = 0;
  int high = agrDemand.length - p - 1;
  while (low + 1 < high) {
    int mid = (low + high + 1) / 2;
    sint t = agrDemand[p + mid] + 1;
    sbool res = agrDemand[p] == t;
    if (open(res | res)) {
      high = mid;
    } else {
      low = mid;
    }
  }
  return high;
}
```

Figure A.9: Implementation of search for marginal prices

```
function bool checkBuyingFarmer(BuyingFarmer f) {
  return (f.getBids().length == 4000);
}
function bool checkSellingFarmer(SellingFarmer f) {
  return f.getBids().length == 4000;
}
```

Figure A.10: Functions for sanity checks

```
function void promptComputeMarginal(AuctionResult res) {}
function void openMarginalBids(int k, int base,
                               AuctionResult res, sint[] agrDemand) {}
```

Figure A.11: Functions stubs for easier integration

# Bibliography


[1] A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *LISP and Functional Programming*, pages 96–105, 1988.

[2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[4] H. Abelson, R. Dybvig, C. Haynes, G. Rozas, N. A. IV, D. Friedman, E. Kohlbecker, G. S. Jr., D. Bartley, R. Halstead, D. Oxley, G. Sussman, G. Brooks, C. Hanson, K. Pitman, and M. Wand. Revised$^5$ report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

[5] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53. ACM Press, 2000.

[6] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. L. S. Jr. Project Fortress Community website. `http://www.projectfortress.sun.com`.

[7] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0. `http://research.sun.com/projects/plrg/fortress.pdf`, Mar. 2008.

[8] E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, and S. Ryu. Growing a syntax. Presented at ACM SIGPLAN Foundations of Object-Oriented Languages workshop, 2009.

[9] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow analysis of pointer programs. Technical Report CIS TR 2005-1, Kansas State University, July 2005.

[10] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 91–102, New York, NY, USA, 2006. ACM Press.

[11] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, June 14 2007.

[12] M. Barbosa, R. Noad, D. Page, and N. P. Smart. First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160, 2005.

[13] M. Barbosa and D. Page. On the automatic construction of indistinguishable operations. In *IMA Int. Conf.*, pages 233–247, 2005.

[14] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, New York, NY, USA, 2008. ACM Press.

[15] P. Bogetoft, K. Boye, H. Neergaard-Petersen, and K. Nielsen. Reallocating sugar beet contracts: Can sugar production survive in denmark? In *European Review of Agricultural Economics*, volume 34, pages 1–20, 2007.

[16] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigård, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, T. Toft, and M. I. Schwartzbach. Secure multiparty computation goes live. In *Proc. of Financial Cryptography*. Springer-Verlag, 2009.

[17] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications. Technical Report RS-05-18, BRICS, June 2005. 37 pp.

[18] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Proc. of Financial Cryptography*, volume 4107 of *LNCS*. Springer-Verlag, 2006.

[19] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Proc. of Financial Cryptography*. Springer-Verlag, 2006.

[20] C. Brabrand and M. I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming Journal (SCP)*, 68(1):2–20, 2007.

[21] G. Bracha. Executable grammars in newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[22] M. Bravenboer, R. D. Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05*. Springer Verlag, 2005.

[23] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[24] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 136–145, 2001.

[25] L. Cardelli and F. Matthes. Extensible syntax with lexical scoping. Technical report, Research Report 121, Digital SRC, 1994.

[26] N. Chomsky. Three models for the description of language. *IEEE Transactions*, 2(3), Sept. 1956.

[27] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *J. Theoretical Computer Science*, 59(3):1–14, Jan. 2004.

[28] W. Clinger and J. Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM.

[29] The COLA Programming Language. `http://piumarta.com/software/cola/`.

[30] R. Cramer and I. Damgård. Multiparty computation, an introduction, 2004.

[31] K. Crary, R. Harper, F. Pfenning, B. C. Pierce, S. Weirich, and S. Zdancewic. Manifest security for distributed information. White paper, http://www.cis.upenn.edu/~ bcpierce/papers/incertproposal06.pdf, Mar. 2006.

[32] R. Culpepper and M. Felleisen. Taming macros. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 225–243. Springer, 2004.

[33] R. Culpepper, S. Owens, and M. Flatt. Syntactic abstraction in component interfaces. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2005.

[34] I. Damgård, M. Geisler, M. Krøigård, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography*, 2009.

[35] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. Theory of Cryptography Conference*, volume 3876 of *LNCS*, pages 285–304. Springer-Verlag, May 2006.

[36] D. E. R. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[37] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, Boston, MA, USA, 1982.

[38] D. E. R. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[39] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992.

[40] M. Flatt. Composable and compilable macros: you want it when? In *ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

[41] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[42] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Sci. Comput. Program.*, 50(1-3):73–99, 2004.

[43] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[44] O. Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.

[45] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. CSFW 15*, 2002.

[46] R. Grimm. Practical packrat parsing. Technical report, New York University, 2004.

[47] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[48] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[49] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[50] K. E. Iverson. *A Programming Language*. Wiley, 1962.

[51] T. Jakobsen and S. From. Secure multi-party computation on integers. Master's thesis, Department of Computer Science, DAIMI, University of Aarhus, Denmark, July 2005.

[52] S. C. Johnson. YACC: Yet Another Compiler-Compiler. *Unix Programmer's Manual*, 2b, 1979.

[53] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 303–310. ACM Press, 1991.

[54] G. L. S. Jr. *Common Lisp the Language*. Digital Press, 1984.

[55] G. L. S. Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[56] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. International Cryptology Conference on Advances in Cryptology*, volume 1109 of *LNCS*, pages 104–113, London, UK, 1996. Springer-Verlag.

[57] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[58] E. E. Kohlbecker and M. Wand. Macros-by-example. In *POPL '87: Proceedings of the 14th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–84, New York, NY, USA, 1987. ACM.

[59] B. Köpf and D. A. Basin. Timing-sensitive information flow analysis for synchronous systems. In *Proc. European Symp. on Research in Computer Security*, pages 243–262, 2006.

[60] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*, 36(3):81–92, 2001.

[61] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[62] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-Party Computation System. In *Proc. of USENIX Security Symposium*, pages 287–302, 2004.

[63] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *Proc. of the ASIAN Symposium on Programming Languages and Systems*, volume 3303 of *LNCS*, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[64] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society Press.

[65] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *In Cryptology ePrint Archive, Report 2005/368*, 2005.

[66] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[67] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, 2006.

[68] J. D. Nielsen. A Domai-specific Programming Language for Secure Multiparty Computation - PhD Progress Report, June 2007.

[69] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In M. W. Hicks, editor, *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 21–30, New York, NY, USA, 2007. ACM Press.

[70] J. D. Nielsen and M. I. Schwartzbach. The SMCL Language Specification. Technical Report RS-07-9, BRICS, Apr. 2007.

[71] J. D. Nielsen and M. I. Schwartzbach. SMCL Security Guaranteed. *Submitted to ACM Transactions on Programming Languages and Systems*, 2009.

[72] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[73] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[74] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[75] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, 2002.

[76] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[77] J. Riehl. Assimilating metaborg:: embedding language tools in languages. In *International Conference on Generative Programming and Component Engineering*, pages 21–28, New York, NY, USA, 2006. ACM.

[78] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21, 2003.

[79] A. Sabelfeld and A. Myers. A model for delimited information release. In *Proc. of the International Symposium on Software Security*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.

[80] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symposium on Programming*, pages 40–58, 1999.

[81] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, page 200, Washington, DC, USA, July 2000. IEEE Computer Society Press.

[82] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society Press.

[83] Sablecc. `http://sablecc.org`.

[84] Scheme frequently asked questions. `http://community.schemewiki.org/?scheme-faq-macros`.

[85] C. G. Seaton. A programming language where the syntax and semantics are mutable at runtime. Master's thesis, Department of Computer Science, University of Bristol, United Kingdom, May 2007.

[86] A. Shamir. How to share a secret. *Comm. of the ACM*, 22(11):612–613, 1979.

[87] M. C. Silaghi. SMC: Secure Multiparty Computation language, 2004. `http://www.cs.fit.edu/msilaghi/SMC/tutorial.html`.

[88] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, New York, NY, 1998.

[89] M. Sperber, W. Clinger, R. K. Dybvig, M. Flatt, A. van Straaten, R. Kelsey, and J. R. (Editors). Revised$^6$ report of the algorithmic language Scheme, Sept. 2007. Available at `http://www.r6rs.org`.

[90] Squeak smalltalk. `http://www.squeak.org/`.

[91] D. R. Stinson. *Cryptography Theory and Practice, Third Edition*. Chapman & Hall/CRC, 2006.

[92] The Coq Proof Assistant. `http://pauillac.inria.fr/coq/`.

[93] The HOL Theorem Prover. `http://hol.sourceforge.net/`.

[94] The Isabelle Theorem Prover. `http://isabelle.in.tum.de/`.

[95] The Unicode Consortium. *The Unicode Standard, Version 5.0*. Addison-Wesley, 2006.

[96] T. Toft. Progress report - Secure Integer Computation with Applications in Economics., July 2005.

[97] T. K. Tolstrup and F. Nielson. Analyzing for Absence of Timing Leaks in VHDL. In D. Gollmann and J. Jürjens, editors, *Proc. International Workshop on Issues in the Theory of Security*, Mar. 2006.

[98] T. K. Tolstrup, F. Nielson, and H. R. Nielson. Information Flow Analysis for VHDL. In V. E. Malyshkin, editor, *Proc. International Conference on Parallel Computing Technologies*, volume 3606 of *LNCS*, pages 79–98. Springer-Verlag, Sept. 2005.

[99] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[100] E. Visser and E. Visser. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*, pages 299–315. Springer-Verlag, 2002.

[101] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. of Theory and Practice of Software Development*, pages 607–621, 1997.

[102] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[103] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[104] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–215, 1999.

[105] A. Warth and I. Piumarta. Ometa: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

[106] A. Warth and I. Piumarta. Ometa: an object-oriented language for pattern matching. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2007. ACM Press.

[107] E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

[108] E. V. Wyk, O. D. Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction, volume 2304 of LNCS*, pages 128–142. Springer-Verlag, 2002.

[109] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 160–164, 1982.

[110] S. Zdancewic. A type system for robust declassification. In *Proc. of the Mathematical Foundations of Programming Semantics*, Mar. 2003.

[111] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, June 2001.

[112] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. Technical Report 2001-1846, Cornell University, 2001.