National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Compiler backend implementation for Noisy

Experimenting on a new language for embedded systems

Diploma Thesis

of

ANGELOS PLEVRIS

Supervisor: Dimitrios Soudris
Professor

Athens, May 2022

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Compiler backend implementation for Noisy

Experimenting on a new language for embedded systems

Diploma Thesis

of

ANGELOS PLEVRIS

Supervisor: Dimitrios Soudris
Professor

Approved by the examination committee on 6th May 2022.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Dimitrios Soudris	Panagiotis Tsanakas	Sotirios Xydis
Professor	Professor	Professor

Athens, May 2022

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

. . . . . .
Angelos Plevris

6th May 2022

Abstract

With this diploma thesis we develop a compiler backend for a subset of the programming language Noisy. Noisy is a new high level programming language designed for embedded systems and real-time computing platforms. Noisy aims to provide embedded systems designers and users with a new set of tools to simplify and also boost their work. Our current work implements the basic computational core of Noisy and should act as a solid foundation for the further development of this language. In this thesis we used the already implemented lexer and parser of Noisy and we created a semantic analyzer as well as a code generator. For the code generation we used the LLVM toolchain which enables us to generate assembly code for many different target architectures. Also, since Noisy aims to implement the CSP (Communicating sequential processes) model, we experimented on its implementation using the LLVM's coroutines. Finally, we wrote and tested a variety of programs in Noisy in order to evaluate and measure our compiler's performance as well as the performance of generated code.

Keywords

LLVM, code generation, CSP, coroutines, compilers, embedded systems.

to my parents

Acknowledgements

First of all, I would like to thank professor Dimitrios Soudris for giving me the opportunity to work in his laboratory. I would also like to thank professor Phillip Stanley-Marbell for his guidance and constant belief in my abilities as well as express my gratitude for giving me the opportunity to work in such a fascinating project of his. Moreover, I would like to thank post doctorate student Vasilis Tsoutsouras. This project would not have been possible if it was not for his constant guidance, cooperation and helpful insight. Also, I would like to thank my friends who have been very supportive all those years and especially this period of my life which has also been a very difficult period for everyone due to COVID. Finally, I would like to thank my parents Pavlos and Vasiliki, who have never stopped caring or providing for me, allowing me to accomplish all my plans, academic and not.

Athens, May 2022

Angelos Plevris

Table of Contents

List of Figures

List of Images

List of Tables

Chapter 1

Εκτεταμένη περίληψη

Εισαγωγή

Σήμερα, τα ενσωματωμένα συστήματα και οι ενσωματωμένες υπολογιστικές συσκευές παίζουν όλο και μεγαλύτερο ρόλο στη ζωή μας. Ως ενσωματωμένο σύστημα ορίζουμε ένα υπολογιστικό σύστημα (δηλαδή έναν επεξεργαστή, μια μνήμη και περιφερειακές συσκευές εισόδου / εξόδου), το οποίο έχει μια συγκεκριμένη λειτουργία μέσα σε ένα ευρύτερο μηχανικό ή ηλεκτρονικό σύστημα [1, 2]. Τέτοια συστήματα χρησιμοποιούμε καθημερινά στα αυτοκίνητα, στα κινητά, στα αεροπλάνα και σε άλλους τομεις της καθημερινής ζωής και της βιομηχανίας. Κύριο χαρακτηριστικό των ενσωματωμένων συστημάτων είναι ότι αφενός λειτουργούν σε πραγματικό χρόνο (δηλαδή πραγματοποιούν υπολογισμούς που πρέπει να παραγάγουν αποτέλεσμα εντός κάποιου ορισμένου χρονικού πλαισίου) και αφετέρου ότι αλληλεπιδρούν με το περιβάλλον τους, κυρίως χρησιμοποιώντας αισθητήρες.

Τα ενσωματωμένα συστήματα αλληλεπιδρούν με το περιβάλλον τους και πραγματοποιούν υπολογισμούς που αναφέρονται σε φυσικά μεγέθη και προβλήματα του πραγματικού κόσμου. Ταυτόχρονα, όμως, οι μετρήσεις των αισθητήρων που χρησιμοποιούνται για αυτή την αλληλεπίδραση μπορεί να περιέχουν αβεβαιότητα και σφάλματα, είτε από θόρυβο των μετρήσεων, είτε από παρεμβολές του περιβάλλοντος, απώλεια δεδομένων κ.α. Επίσης, οι φυσικοι περιορισμοί που αφορούν το εκάστοτε πρόβλημα που καλείται να λύσει ένα ενσωματωμένο σύστημα, πρέπει και αυτοί να λαμβάνονται υπόψη. Αυτό έχει σαν αποτέλεσμα τα ενσωματωμένα να εμφανίζουν αρκετές ιδιαιτερότητες ως κλάδος και να δημιουργείται η ανάγκη για αρκετές διαφορετικές λύσεις ώστε να βελτιώνονται σε όρους απόδοσης καθώς και ενεργειακής κατανάλωσης, λύσεις που είτε αφορούν καλύτερο υλικό (επεξεργαστές, μνήμες, διασύνδεση κ.α.), είτε καλύτερο λογισμικό (αλγόριθμοι, βελτιώσεις στις γλώσσες προγραμματισμού κ.α.).

Παρά τις συνεχείς αλλαγές και βελτιώσεις που γίνονται στον τομέα των ενσωματωμένων συστημάτων, η γλώσσα προγραμματισμού C παραμένει μία σταθερά στον τομέα αυτόν, όντας μία από τις πιο παλιές, ευρέως υποστηριζόμενη, σταθερή και γρήγορη γλώσσα. Όμως, ακριβώς επειδή η C είναι αρκετά παλιά δεν προσφέρει νεότερα προγραμματιστικά ιδιώματα και εργαλεία που θα μπορούσαν να είναι χρήσιμα για τους σχεδιαστές και χρήστες των ενσωματωμένων συστημάτων. Ακόμη, η πρόσθεση κάποιων νέων χαρακτηριστικών στη γλώσσα C θα ήταν μια δύσκολη διαδικασία που θα απαιτούσε να τροποποιηθεί ο μεταγλωττιστής της ή να γραφτούν καινούριες βιβλιοθήκες. Για αυτό τον λόγο έχει δημιουργηθεί η ανάγκη για ένα προγραμματιστικό μοντέλο στοχευμένο στα ενσωματωμένα συστήματα.

Η Noisy είναι μια καινούρια γλώσσα προγραμματισμού που χρησιμοποιεί ιδιώματα και προγραμματιστικές κατασκευές χρήσιμες για τον τομέα των ενσωματωμένων συστημάτων και σκοπεύει στο να παρέχει τα απαραίτητα εργαλεία για την ανάπτυξη και χρήση τέτοιων συστημάτων. Στην παρούσα εργασία, παρουσιάζουμε τα βασικά χαρακτηριστικά αυτής της γλώσσας καθώς και την υλοποίηση του οπίσθιου τμήματος του μεταγλωττιστή της. Για την υλοποίηση του μεταγλωττιστή επιλέξαμε μόνο το βασικό υπολογιστικό υποσύνολο της γλώσσας και επομένως πολλά χαρακτηριστικά της γλώσσας μένουν να υλοποιηθούν σε μελλοντική εργασία.

Η γλώσσα Noisy

Η γλώσσα Noisy, σχεδιασμένη από τον καθηγητή Phillip Stanley-Marbell, είναι μία γλώσσα προγραμματισμού γενικού σκοπού που παρέχει εργαλεία για την επεξεργασία σημάτων από τον φυσικό κόσμο. Σκοπός της γλώσσας είναι η διερεύνηση του πώς η πληροφορία για φυσικά σήματα και μεγέθη μπορεί να αναπαρασταθεί και να αξιοποιηθεί σε επίπεδο γλώσσας προγραμματισμού και μεταγλωττιστή, ώστε να βελτιωθεί η ασφάλεια, η απόδοση και η αποδοτικότητα των προγραμμάτων ενσωματωμένων συστημάτων.

Η γλώσσα Noisy είναι προστακτική και μεταγλωττίσιμη, δηλαδή χρησιμοποιεί μεταγλωττιστή για να μπορεί να μεταφραστεί σε κώδικα μηχανής. Ο μεταγλωττιστής διαβάζει ένα πρόγραμμα γραμμένο σε Noisy και επιστρέφει ένα σημασιολογικά ισοδύναμο πρόγραμμα γραμμένο σε γλώσσα μηχανής. Η Noisy παρέχει επίσης, αυστηρό σύστημα τύπων το οποίο ελέγχεται στατικά κατά την μεταγλώττιση. Τέλος, η Noisy παρέχει υποστήριξη στο μοντέλο Επικοινωνουσών Ακολουθιακών Διεργασιών (Communicating Sequential Processes - CSP) [3] το οποίο επιτρέπει την εύκολη υλοποίηση χαρακτηριστικών ταυτοχρονισμού και παραλληλίας, ενώ οι εντολές της γλώσσας είναι έντονα επηρεασμένες από τις φρουρούμενες εντολές του Dijkstra [4].

Ένα πρόγραμμα γραμμένο σε Noisy αποτελείται από τουλάχιστον μια ενότητα (module) και από μια συλλογή γεννητριών ονομάτων (name-generators). Στην Noisy χρησιμοποιούμε τον όρο ¨γεννήτριες ονομάτων' για να αναφερθούμε σε συναρτήσεις (functions), καθώς και σε κανάλια επικοινωνίας (channels), τα οποία χρησιμοποιούνται για την υλοποίηση του CSP μοντέλου. Κάθε ενότητα αποτελείται από δηλώσεις γεννητριών ονομάτων, δηλώσεις σταθερών, και δηλώσεις ονομάτων τύπων. Για να μεταγλωττιστεί ένα πρόγραμμα χρειάζεται να έχει μία ενότητα και τις υλοποιήσεις των συναρτήσεων του. Ένα πρόγραμμα μπορεί να περιέχει και τοπικές γεννήτριες ονομάτων.

Στη γλώσσα Noisy υπάρχουν δύο είδη αναγνωριστικών (identifiers): οι μεταβλητές (variables) και τα κανάλια (channels). Οι μεταβλητές αναπαριστούν θέσεις αποθήκευσης στη μνήμη που έχουν συγκεκριμένο όνομα. Τα κανάλια αποτελούν μηχανισμό επικοινωνίας μεταξύ δύο ακολουθιακών τμημάτων κώδικα που θέλουν να ανταλλάξουν δεδομένα. Τα κανάλια λειτουργούν μονόδρομα (δηλαδή ο αποστολέας που χρησιμοποιεί ένα κανάλι δεν μπορεί να χρησιμοποιήσει το ίδιο κανάλι για να λάβει δεδομένα), έχουν τύπο (δηλαδή μπορούν να αποστέλλονται δεδομένα μόνο ενός τύπου από το συγκεκριμένο κανάλι), είναι σύγχρονα (δηλαδή μια αποστολή δεδομένων ολοκληρώνεται όταν και ο παραλήπτης τα παραλαμβάνει), και δεν χρησιμοποιούν προσωρινή αποθήκευση για την υλοποίηση τους (unbuffered), που σημαίνει ότι δεν μπορεί ο αποστολέας να στείλει και άλλα δεδομένα, εάν δεν έχουν παραληφθεί από

τον παραλήπτη τα πρώτα δεδομένα που έχουν σταλεί.

Οι κανόνες εμβέλειας ονομάτων στη Noisy ορίζονται παρόμοια με άλλες γλώσσες προγραμματισμού και παρουσιάζονται στη γραμματική της γλώσσας. Συνήθως μια καινούρια εμβέλεια (scope) ορίζεται σε καινούριες ενότητες, σε Αφηρημένους Τύπους Δεδομένων και σε εντολές που έχουν { και }.

Η γλώσσα Noisy παρέχει αυστηρό σύστημα τύπων και τα προγράμματα ελέγχονται για λάθη τύπων κατά τη μεταγλώττιση. Στη γλώσσα, υπάρχουν οι απλοί τύπου (αριθμητικοί και μη) και οι σύνθετοι τύποι που αφορούν συλλογές δεδομένων. Οι απλοί αριθμητικοί τύποι της γλωσσας είναι οι ακέραιοι, οι φυσικοί, οι πραγματικοί κινητής υποδιαστολής και σταθερής υποδιαστολής. Για τους ακέραιους, τους φυσικούς και τους πραγματικούς κινητής υποδιαστολής υποστηρίζονται διαφορετικά μεγέθη από 4 bit έως 128 bit. Οι μη αριθμητικοί απλοί τύποι είναι οι δυαδικές λογικές τιμές καθώς και οι συμβολοσειρές. Οι σύνθετοι τύποι που υποστηρίζονται είναι οι στατικοί πίνακες, οι δυναμικές λίστες, τα σύνολα, τα Ευκλείδεια διανύσματα, οι πλειάδες και οι αφηρημένοι τύποι δεδομένων. Για τους διάφορους τύπους δεδομένων ορίζονται και διαφορετικοί τελεστές με διαφορετικές ιδιότητες και σημασιολογία.

Στην γλώσσα υπάρχουν εκφράσεις, εντολές ανάθεσης, εντολές επιλογής και εντολές επανάληψης. Οι εκφράσεις είναι συντακτικές οντότητες που αποτιμώνται, ώστε να καθοριστεί η τιμή τους. Οι εντολές ανάθεσης αποθηκεύουν μια τιμή σε μία ή περισσότερες μεταβλητες. Οι εντολές επιλογής (**match** / **matchseq**) περιέχουν φρουρούμενες εντολές οι οποίες εκτελούνται μόνο εάν η συνθήκη του εκάστοτε φρουρού αληθεύει. Παρομοίως, η εντολή επανάληψης **iterate** εκτελείται επαναληπτικά όσο αληθεύει η συνθήκη φρουρός. Τέλος, η εντολή επανάληψης **sequence** λειτουργεί παρόμοια με την εντολή *for* που συναντάμε σε άλλες γλώσσες.

Τέλος, στον αρχικό σχεδιασμό της γλώσσας Noisy, υπάρχει και η διασύνδεση της με τη γλώσσα Newton. Η γλώσσα Newton [5] απευθύνεται σε κατασκευαστές αισθητήρων και περιφερειακών συσκευών ενσωματωμένων και επιτρέπει τη δήλωση αναλλοίωτων και περιορισμών μεταξύ ροών δεδομένων που παρέχονται απο αισθητήρες. Αυτή η πληροφορία μπορεί να χρησιμοποιηθεί κατά τη μεταγλώττιση ή κατά την εκτέλεση ενός προγράμματος, για να βελτιώσει την απόδοση του. Στόχος των δύο αυτών γλωσσών είναι να λειτουργούν συμπληρωματικά, ώστε να γεφυρώνουν το χάσμα μεταξύ των διεπαφών που χρησιμοποιούν οι κατασκευαστές του υλικού και των διεπαφών και προγραμματιστικών ιδιωμάτων που χρησιμοποιούν οι χρήστες του υλικού, δηλαδή οι προγραμματιστές. Για αυτόν τον λόγο, οι δύο αυτές γλώσσες χρησιμοποιούν κοινή υποδομή στους μεταγλωττιστές τους και η διασύνδεση των δύο θα μπορούσε να μελετηθεί σε μελλοντική εργασία.

Σχεδιασμός και υλοποίηση

Πριν την παρούσα εργασία, η υλοποίηση του μεταγλωττιστή της Noisy ήταν σε ένα πρώιμο στάδιο. Συγκεκριμένα, είχε υλοποιηθεί ο λεκτικός αναλυτής (lexer), ο οποίος αναγνωρίζει τις λεκτικές μονάδες ενός προγράμματος και ο συντακτικός αναλυτής (parser), ο οποίος από ένα δοθέν πρόγραμμα δημιουργεί στη μνήμη, μια ισοδύναμη εσωτερική αναπαράσταση του προγράμματος, η οποία ονομάζεται Αφηρημένο Συντακτικό Δέντρο (Abstract Syntax Tree - AST). Ο συντακτικός αναλυτής χρησιμοποιεί την τεχνική της καθοδικής συντακτικής ανάλυσης (top-down parsing) , καθώς η γλώσσα ανήκει στην κατηγορία γλωσσών που ονομάζεται LL(1). Η

θεωρία και η υλοποίηση συντακτικών αναλυτών για γλώσσες LL(1) είναι ευρέως διαδεδομένη στη βιβλιογραφία.

Η δικιά μας υλοποίηση ξεκινάει από την προηγούμενη υλοποίηση του μεταγλωττιστή και προσθέτει σε αυτόν ένα στάδιο σημασιολογικής ανάλυσης το οποίο είναι υπεύθυνο, κυρίως, για τους ελέγχους των τύπων των προγραμμάτων, και ένα στάδιο παραγωγής κώδικα το οποίο διασχίζει το Αφηρημένο Συντακτικό Δέντρο ενός προγράμματος και παράγει ισοδύναμο κώδικα μηχανής.

Σημασιολογική Ανάλυση

Αρχικά, στην υλοποίηση μας πραγματοποιούμε ελέγχους τύπων για όλους τους απλούς αριθμητικούς τύπους και λογικές τιμές, εκτός από τους πραγματικούς αριθμούς σταθερής υποδιαστολής. Από σύνθετους τύπους υποστηρίζουμε μόνο τους πίνακες σταθερού μεγέθους. Προκειμένου να πραγματοποιηθεί η σημασιολογική ανάλυση, χρησιμοποιούμε τον πίνακα συμβολών, που έιχε υλοποιηθεί από την προηγούμενη εργασία. Στον πίνακα συμβόλων, αποθηκεύονται τα ονόματα των προσδιοριστών, η εμβέλεια στην οποία ανήκουν, καθώς και ο τύπος τους σε μορφή δέντρου. Οι τύποι στη Noisy, εκτός από τον τύπο που έχει σχέση με την αναπαράσταση τους στη μνήμη, έχουν και προσδιοριστικά τύπων που αφορούν τις διαστάσεις τους (ως φυσικά μεγέθη), μονάδες μέτρησης κ.α. Στην υλοποίηση μας οι έλεγχοι τύπων αφορούν μόνο την αναπαράσταση στη μνήμη και όχι τις διαστάσεις ή τις μονάδες μέτρησης. Προκειμένου να ελέγξουμε τον τύπο κάποιου προσδιοριστή, η διαδικασία που απαιτείται είναι να τον βρούμε στον πίνακα συμβόλων και έπειτα να διασχίσουμε το δέντρο τύπου του. Παρ' όλα αυτά, επειδή ελέγχουμε μόνο τον τύπο που αφορά την αναπαράσταση στη μνήμη, για να απλοποιήσουμε τη διαδικασία ελέγχου, για κάθε προσδιοριστή που διασχίζουμε το δέντρο τύπου του, αποθηκεύουμε τον τύπο του σε μια πιο απλοποιημένη μορφή. Αυτή η απλοποιημένη μορφή, είναι μια απαρίθμηση που δημιουργήσαμε εμείς και την χρησιμοποιεί εσωτερικά ο μεταγλωττιστής για να κάνει τους ελέγχους τύπων, και ονομάζεται NoisyBasicType. Επίσης, δημιουργήσαμε άλλη μία δομή δεδομένων, την οποία ονομάσαμε NoisyType. Αυτή η δομή περιέχει το πεδίο NoisyBasicType καθώς και άλλα πεδία που αφορούν ειδικές περιπτώσεις όπως θα δούμε παρακάτω.

Το πεδίο NoisyBasicType παίρνει πολλές διαφορετικές τιμές, από τις οποίες, κάθε μία αντιστοιχεί και σε έναν απλό τύπο δεδομένων. Πέρα από αυτές τις τιμές παίρνει και ειδικές τιμές που κωδικοποιούν η μεταβλητή να είναι πίνακας, γεννήτρια ονόματος, η ειδική τιμή **nil**, ακέραια ή πραγματική σταθερά, ή λάθος τύπου.

Στην περίπτωση που μία τιμή έχει τύπο πίνακα το πεδίο NoisyBasicType παίρνει την τιμή noisyArrayType και επίσης απαιτείται να αποθηκεύσουμε στη δομή NoisyType επιπλέον πληροφορία που αφορά τον τύπο των στοιχείων του πίνακα, το πλήθος των διαστάσεων του και το μέγεθος κάθε διάστασης.

Στην περίπτωση που μια τιμή είναι σταθερά, αποθηκεύουμε στο πεδίο NoisyBasicType ότι είναι σταθερά, καθώς στον έλεγχο τύπων οι σταθερές πρέπει να ταιριάζουν με οποιοδήποτε ίδιο τύπο ανεξαρτήτως μεγέθους. Για παράδειγμα, όταν προσθέτουμε δύο μεταβλητές a,b και η μία έχει τύπο int32 και η άλλη int16, ο μεταγλωττιστής θεωρεί ότι αυτή η πράξη είναι σφάλμα τύπου καθώς οι δύο μεταβλητές έχουν διαφορετικό μέγεθος. Στην περίπτωση όμως πρόσθεσης

με σταθερά, ο μεταγλωττιστής πρέπει να θεωρεί αποδεκτές όλες τις πράξεις εφόσον η σταθερά είναι του ίδιου τύπου ανεξαρτήτως μεγέθους. Για αυτόν τον λόγο επιλέξαμε να αναπαριστούμε με ξεχωριστό τύπο τις σταθερές και επίσης υλοποιήσαμε ένα μικρό σύστημα συμπερασμού τύπων για σταθερές, ώστε σε μια έκφραση οι σταθερές να λαμβάνουν τον κατάλληλο τύπο ώστε να θεωρείται ορθή η έκφραση.

Επίσης υλοποιήσαμε και έναν μηχανισμό για πρότυπα τύπων (type templates) ο οποίος επιτρέπει στον προγραμματιστή να δηλώνει συναρτήσεις που χρησιμοποιούν πρότυπα τύπων και όταν θέλει να χρησιμοποιήσει αυτές τις συναρτήσεις, μπορεί να δώσει συγκεκριμένο τύπο στα πρότυπα αυτά και να εξειδικεύσει τη συγκεκριμένη συνάρτηση για τον τύπο που θέλει. Αυτή η εξειδίκευση και ο έλεγχος τύπων γίνεται κατά τη διάρκεια της μεταγλώττισης και όχι κατά τη διάρκεια της εκτέλεσης του προγράμματος. Για να υλοποιήσουμε αυτόν τον μηχανισμό, για κάθε συνάρτηση η οποία χρησιμοποιεί πρότυπα τύπων δημιουργούμε ένα αντίγραφο του Αφηρημένου Συντακτικού Δέντρου της και το αποθηκεύουμε στη μνήμη, καθώς και φτιάχνουμε αντίγραφα των μεταβλητών της στον πίνακα συμβόλων. Όταν η συνάρτηση εξειδικεύεται για έναν συγκεκριμένο τύπο, τότε οι τιμές που αφορούν τύπους στο Αφηρημένο Συντακτικό Δέντρο, καθώς και στον πίνακα συμβόλων, αντικαθιστώνται από τον κατάλληλο τύπο που έχει δώσει ο χρήστης για να εξειδικεύσει τη συνάρτηση. Αυτή η διαδικασία γίνεται μόνο στα αντίγραφα, ενώ το αρχικό δέντρο και οι εγγραφές παραμένουν ίδιες ώστε η ίδια συνάρτηση να μπορεί να εξειδικευτεί πολλές φορές μέσα στο ίδιο πρόγραμμα.

Επίσης, πέρα από τις ειδικές περιπτώσεις που αναφέραμε, η σημασιολογική ανάλυση ορίζει κανόνες τύπων για όλες τις εκφράσεις και εντολές της γλώσσας. Για τις εκφράσεις ορίζουμε ότι μια έκφραση είναι ορθή σημασιολογικά αν τα τελούμενα σε κάθε πράξη έχουν σωστούς τύπους (είτε τον ίδιο για κάποιους τελεστές, είτε κατάλληλο τύπο για άλλους τελεστές). Για τις εντολές ανάθεσης πρέπει το αριστερό μέλος μιας ανάθεσης να έχει τον ίδιο τύπο με το δεξί μέλος (με εξαίρεση την τιμή **nil** που ταιριάζει με όλους τους τύπους). Στις φρουρούμενες εντολές, πρέπει οι εκφράσεις συνθήκης να αποτιμώνται ως δυαδικές λογικές τιμές. Επίσης, υπάρχουν κανόνες που αφορούν την κλήση συναρτήσεων και την επιστροφή τιμών από αυτές. Τέλος, η γλώσσα δίνει τη δυνατότητα για μετατροπές από έναν αριθμητικό τύπο σε άλλον. Σε περίπτωση που η σημασιολογική ανάλυση εντοπίσει κάποιο λάθος τύπου, τότε η μεταγλώττιση του προγράμματος διακόπτεται και εκτυπώνεται στην οθόνη μήνυμα που αναγράφει το λάθος τύπου, καθώς και τη θέση στην οποία εντοπίζεται μέσα στο πρόγραμμα.

Ένα άλλο σημαντικό κομμάτι της υλοποίησης μας είναι και η υλοποίηση των καναλιών επικοινωνίας και των συρρουτίνων (coroutines). Αναφέραμε προηγουμένως ότι στη γλώσσα υπάρχουν οι γεννήτριες ονομάτων, οι οποίες μπορεί να λειτουργούν είτε ως συναρτήσεις είτε ως συρρουτίνες για την υλοποίηση του μοντέλου Επικοινωνουσών Ακολουθιακών Διεργασιών. Στο μοντέλο αυτό, υπάρχουν οι συρρουτίνες που δρουν ως ακολουθιακές διεργασίες, οι οποίες μπορούν να ανταλλάξουν μεταξύ τους δεδομένα μόνο με τη χρήση των καναλιών επικοινωνίας. Το ιδιαίτερο χαρακτηριστικό των συρρουτίνων είναι ότι μπορούν να διακόψουν την εκτέλεση τους, να παραχωρήσουν τη ροή εκτέλεσης σε άλλη συρρουτίνα και έπειτα να ξαναρχίσουν την εκτέλεση τους από το σημείο στο οποίο τη διέκοψαν. Τα σημεία στα οποία παραχωρείται ο έλεγχος σε άλλη συρρουτίνα, είναι τα σημεία στα οποία γίνεται επικοινωνία μέσω ενός καναλιού (είτε στο γράψιμο, είτε στο διάβασμα από το κανάλι). Για τη σημασιολογική ανάλυση ο έλεγχος που απαιτείται, είναι αρχικά, να αναγνωρίζει ο μεταγλωττιστής ότι μία γεννήτρια

ονόματος λειτουργεί ως συρρουτίνα και όχι ως συνάρτηση. Αυτό επιτυγχάνεται με τον έλεγχο του κατά πόσο, στο σώμα του κώδικα της γεννήτριας ή κατά την κλήση της, χρησιμοποιείται σημειολογία που αφορά διαχείριση καναλιών (π.χ. οι τελεστές $->$, $<-=$). Τέλος, η σημασιολογική ανάλυση ελέγχει τους τύπους των καναλιών και των πράξεων που έχουν σχέση με κανάλια. Τέλος, με την παρούσα υλοποίηση, μία συρρουτίνα μπορεί να έχει μόνο μέχρι ένα κανάλι εισόδου και μέχρι ένα κανάλι εξόδου.

Παραγωγή Κώδικα

Ο παραγωγός κώδικα είναι το κομμάτι εκείνο του μεταγλωττιστή το οποίο είναι υπεύθυνο για να μετατρέπει το Αφηρημένο Συντακτικό Δέντρο που έχει δημιουργήσει ο μεταγλωττιστής σε προηγούμενα στάδια, σε μια σημασιολογικά ισοδύναμη εκτελέσιμη μορφή. Για τον παραγωγό κώδικα χρησιμοποιήσαμε τα εργαλεία και τη μεθοδολογία που μας παρέχει το LLVM (Low Level Virtual Machine). Συγκεκριμένα, στην παραγωγή κώδικα κάθε κόμβος του Αφηρημένου Συντακτικού Δέντρου μεταφράζεται σε κάποια ειδική τιμή LLVM IR (Intermediate Representation). Το LLVM IR είναι μια γλώσσα ενδιάμεσης περιγραφής χαμηλού επιπέδου, ανεξάρτητη από τη γλώσσα-πηγή και την αρχιτεκτονική-στόχο, η οποία χρησιμοποιεί αυστηρό σύστημα τύπων και το μοντέλο Απλής Στατικής Ανάθεσης (Single Static Assignment - SSA). Το πλεονέκτημα που μας προσφέρει η χρήση του LLVM είναι ότι εμείς μπορούμε να γράψουμε τον παραγωγό κώδικα που παράγει τιμές LLVM IR και έπειτα κάνοντας χρήση των εργαλείων του LLVM να δημιουργήσουμε εκτελέσιμα για ένα πολύ μεγάλος πλήθος αρχιτεκτονικών και επεξεργαστών, οι οποίες υποστηρίζονται από το ίδιο το LLVM χωρίς να χρειαστεί εμείς να γράψουμε κάποιον κώδικα μηχανής. Τέλος, άλλο ένα πλεονέκτημα είναι ότι το LLVM παρέχει αυτόματες και έτοιμες βελτιστοποιήσεις κώδικα που μπορούμε να χρησιμοποιήσουμε. Για την υλοποίηση μας χρησιμοποιήσαμε την διεπαφή της βιβλιοθήκης του LLVM για τη γλώσσα C [6], στην οποία είναι γραμμένη και η υπόλοιπη υλοποίηση του μεταγλωττιστή μας.

Ένα πρόγραμμα γραμμένο σε LLVM IR αποτελείται από ένα Module, το οποίο αποτελείται από δηλώσεις σταθερών τιμών και συναρτήσεων (functions). Οι γεννήτριες ονομάτων που συναντάμε στη Noisy παράγουν συναρτήσεις, ενώ δέχονται ειδικό χειρισμό όταν λειτουργούν ως συρρουτίνες. Οι συναρτήσεις με τη σειρά τους αποτελούνται από Basic Blocks τα οποία περιέχουν όλες τις υπόλοιπες εντολές σε LLVM και συνδέονται μεταξύ τους με εντολές άλματος.

Όπως αναφέραμε, ο παραγωγός κώδικα διασχίζει το Αφηρημένο Συντακτικό Δέντρο ενός προγράμματος, χρησιμοποιεί τη σημασιολογική πληροφορία που βρίσκεται αποθηκευμένη στον πίνακα συμβόλων, και με βάση αυτή δημιουργεί εντολές σε LLVM IR. Ο τρόπος που μεταφράζεται κάθε εντολή και έκφραση της Noisy σε εντολές του LLVM IR ποικίλει, αλλά όλη μας η υλοποίηση χρησιμοποιεί τη βασική βιβλιοθήκη του LLVM. Πιο συγκεκριμένα, κάθε έκφραση της γλώσσας αποτιμάται σαν μία τελική τιμή, που ονομάζεται LLVM IR Value. Επίσης κάθε έκφραση έχει και τον δικό της τύπο (LLVM IR Type). Αξίζει να επισημάνουμε ότι σε μια έκφραση, επειδή ακριβώς το LLVM χρησιμοποιεί SSA μορφη, κάθε πράξη παράγει μια μόνο εντολή η οποία εκτελεί αυτή την πράξη, σύμφωνα με την προτεραιότητα των πράξεων της γλώσσας (π.χ. η πρόσθεση 3 αριθμών θα γεννήσει μια εντολή πρόσθεσης του πρώτου και του δεύτερου και μια εντολή πρόσθεσης του πρώτου αποτελέσματος με τον τρίτο αριθμό).

Οι εντολές ανάθεσης της γλώσσας παράγουν εντολές αποθήκευσης στη μνήμη. Οι εντολές επιστροφής ελέγχου παράγουν εντολές επιστροφής, που παραχωρούν τη ροή ελέγχου στην καλούσα συνάρτηση. Οι εντολές που αλλάζουν τον έλεγχο ροής δημιουργούν νέα Basic Blocks και τα συνδέουν κατάλληλα μεταξύ τους, χρησιμοποιώντας εντολές άλματος υπό συνθήκη, είτε χωρίς συνθήκη. Για κάθε εντολή που αλλάζει τη ροή ελέγχου αντιστοιχεί και διαφορετικό διάγραμμα ελέγχου που χρησιμοποιεί τα Basic Blocks.

Ένα ενδιαφέρον κομμάτι της υλοποίησης μας είναι ο χειρισμός των πινάκων. Οι πίνακες έχουν σταθερό μέγεθος το οποίο είναι γνωστό κατά τη διάρκεια μεταγλώττισης και αποθηκεύονται στη στοίβα του υπολογιστή. Οι πίνακες μπορούν να περνούν ως ορίσματα σε συναρτήσεις ή να επιστρέφονται από μια συνάρτηση. Όμως, επειδή στη Noisy δεν υπάρχουν δείκτες, έχουμε επιλέξει να περνάμε τους πίνακες ¨κατά τιμή' στο πέρασμα παραμέτρων (δηλαδή να αντιγράφουμε όλα τα στοιχεία του πίνακα, αντί να περνάμε έναν δείκτη σε αυτόν τον πίνακα), ενώ για την επιστροφή πινάκων αντιγράφουμε τα περιεχόμενα του πίνακα που επιστρέφουμε σε έναν πίνακα που ανήκει στην καλούσα συνάρτηση. Είμαστε αναγκασμένοι να προβούμε σε αυτήν την υλοποίηση, γιατί οι πίνακες μιας συνάρτησης δεν μπορούν να επιστραφούν στην καλούσα συνάρτηση, καθώς βρίσκονται αποθηκευμένοι στη στοίβα και θα αποδεσμευτούν κατά την επιστροφή.

Ακόμη, μεγάλο κομμάτι της υλοποίησης μας είναι και ό,τι αφορά το μοντέλο Επικοινωνουσών Ακολουθιακών Διεργασιών και τις συρρουτίνες. Είδαμε ότι οι γεννήτριες ονομάτων, λειτουργούν ως συρρουτίνες, όταν χρησιμοποιούνε τα κανάλια επικοινωνίας για να μεταφέρουν δεδομένα από μια συρρουτίνα σε άλλη. Το ιδιαίτερο χαρακτηριστικό των συρρουτίνων είναι ότι, όταν πραγματοποιείται μια μεταφορά δεδομένων από κανάλι διακόπτουν την λειτουργία τους. Προκειμένου να υλοποιήσουμε αυτή τη συμπεριφορά των συρρουτίνων, χρησιμοποιήσαμε μια ειδική βιβλιοθήκη του LLVM που υποστηρίζει αυτή τη λειτουργία και ονομάζεται LLVM Coroutines [7]. Συγκεκριμένα, τα LLVM Coroutines μας δίνουν τη δυνατότητα να γράφουμε συναρτήσεις που παραχωρούν τον έλεγχο ροής σε άλλες συναρτήσεις, αποθηκεύοντας όμως την κατάσταση στην οποία βρίσκονταν κατά τη διακοπή, ώστε αργότερα να μπορούν να συνεχίσουν την εκτέλεση τους από το σημείο στο οποίο την είχαν σταματήσει. Για να επιτύχουμε αυτή τη λειτουργικότητα, χρησιμοποιήσαμε μια σειρά από εσωτερικές συναρτήσεις (intrinsic functions) που μας παρέχει το LLVM που ονομάζονται llvm.coro.begin (η αρχική συνάρτηση που υποδηλώνει την αρχικοποίηση και αρχική κλήση μίας συρρουτίνας), llmv.coro.suspend (η οποία υποδηλώνει ότι μια συρρουτίνα αποθηκεύει την κατάσταση στην οποία βρίσκεται και παραχωρεί τον έλεγχο στην καλούσα συνάρτηση), llmv.coro.resume (η οποία υποδηλώνει ότι μια συρρουτίνα συνεχίζει την εκτέλεση της από εκεί που είχε σταματήσει), llvm.coro.destroy (η οποία υποδηλώνει ότι μια συρρουτίνα τερματίζει την εκτέλεση της και αποδεσμεύεται από τη στοίβα). Ο παραγωγός κώδικα είναι υπεύθυνος για να γεννάει κλήσεις για αυτές τις εσωτερικές συναρτήσεις. Έπειτα το πρόγραμμα βελτιστοποιήσεων του LLVM σαρώνει τον κώδικα που παραγάγαμε και αντικαθιστά τις κλήσεις αυτών των εσωτερικών συναρτήσεων με πραγματικές συναρτήσεις. Η μετατροπή από εσωτερικές σε πραγματικές συναρτήσεις γίνεται με βάση διάφορες στρατηγικές που είναι διαθέσιμες από την ίδια τη βιβλιοθήκη και συγκεκριμένα, εμείς επιλέξαμε τη switch-resume lowering. Η στρατηγική αυτή σημαίνει ότι από μία αρχική συρρουτίνα παράγονται 3 συναρτήσεις: ramp function, resume function, destroy function. Η πρώτη συνάρτηση είναι υπεύθυνη για την αρχικοποίηση του πλαισίου στοίβας της

συρρουτίνας. Η δεύτερη συνάρτηση παίρνει ως όρισμα ένα πλαίσιο στοίβας μιας συρρουτίνας, καλείται όταν καλείται η llvm.coro.resume και είναι υπεύθυνη για την συνέχιση της εκτέλεσης μίας συρρουτίνας από το σημείο που είχε μείνει, και η τελευταία συνάρτηση, είναι υπεύθυνη για την καταστροφή και αποδέσμευση της συρρουτίνας. Η στρατηγική switch-resume lowering επομένως, παράγει τις 3 παραπάνω συναρτήσεις για κάθε συρρουτίνα του προγράμματος μας και ονομάζεται έτσι, επειδή χρησιμοποιείται η δομή ελέγχου switch για την επιλογή του σημείου συνέχειας μίας συρρουτίνας. Με βάση αυτή την υλοποίηση, το κόστος της παραχώρηση του ελέγχου από μία συρρουτίνα σε μία άλλη είναι πρακτικά όμοιο με το κόστος κλήσης μιας οποιαδήποτε συνάρτησης και επομένως, η χρήση συρρουτίνων δεν θα έπρεπε να επιβαρύνει σηματνικά τον χρόνο εκτέλεσης ενός προγράμματος.

Όσον αφορά τα κανάλια επικοινωνίας, υλοποιήσαμε μια πρώιμη μορφή για να ελέγξουμε ότι λειτουργούν όπως θα θέλαμε. Για τα κανάλια επικοινωνίας έχουμε δύο περιπτώσεις, το κανάλι εισόδου μίας συρρουτίνας και το κανάλι εξόδου της. Στο κανάλι εισόδου μίας συρρουτίνας μπορεί να γράψει η καλούσα συνάρτηση, ώστε να της περάσει δεδομένα. Η υλοποίηση του καναλιού γίνεται χρησιμοποιώντας μια θέση μνήμης την οποία περνάμε κατ΄ αναφορά στη συρρουτίνα, ώστε να μπορεί η καλούσα συνάρτηση να αλλάζει το περιεχόμενο και να είναι ορατό από τη συρρουτίνα. Για το κανάλι εξόδου, χρησιμοποιούμε έναν έτοιμο μηχανισμό που μας παρέχει το LLVM που ονομάζεται promises. Τα promises λειτουργούν και αυτά σαν μια θέση αποθήκευσης στην οποία γράφει η συρρουτίνα και μπορεί να έχει πρόσβαση η καλούσα συνάρτηση. Η παρούσα υλοποίηση των καναλιών δεν είναι ολοκληρωμένη, καθώς, ενώ λειτουργεί ορθά για προγράμματα που είναι σημασιολογικά ορθά, σε περίπτωση σφαλμάτων στα κανάλια, το πρόγραμμα είτε τερματίζει απρόοπτα είτε παράγει λανθασμένες τιμές, αντί το σύστημα χρόνου εκτέλεσης να εντοπίζει το σφάλμα και να τυπώνει κατάλληλα μηνύματα.

Συνολικά, η διαδικασία της μεταγλώττισης ενός προγράμματος αποτελείται από τα εξής στάδια. Αρχικά, ο πηγαίος κώδικας του προγράμματος γραμμένος σε Noisy περνάει από το εμπρόσθιο τμήμα του μεταγλωττιστή (λεκτική ανάλυση, συντακτική ανάλυση, σημασιολογική ανάλυση) και παράγονται το Αφηρημένο Συντακτικό Δέντρο και ο πίνακας συμβόλων. Έπειτα το οπίσθιο τμήμα του μεταγλωττιστή παράγει ένα ισοδύναμο πρόγραμμα γραμμένο σε LLVM IR. Προαιρετικά, σε περίπτωση που επιθυμούμε να κάνουμε βελτιστοποιήσεις στον παραγώμενο κώδικα, μπορούμε να περάσουμε το πρόγραμμα μας από τον βελτιστοποιητή (optimiser) που μας παρέχει το LLVM και ονομάζεται opt. Στη συνέχεια, χρησιμοποιούμε τον στατικό μεταγλωττιστή του LLVM ο οποίος ονομάζεται llc και παράγει αντικείμενα αρχεία (object files) από το LLVM IR. Τέλος, τα αρχεία αυτά μπορούν να συνδεθούν μεταξύ τους καθώς και με βιβλιοθήκες της C ώστε να παράγουν το τελικό εκτελέσιμο αρχείο, χρησιμοποιώντας είτε τον linker του gcc, είτε κάποιον άλλον linker. Ο μεταγλωττιστής llc είναι αυτός ο οποίος μας επιτρέπει να επιλέξουμε τον στόχο για τον οποίο θέλουμε να μεταγλωττίσουμε το πρόγραμμα μας, παρέχοντας μας μια σειρά από επιλογές που αφορούν την αρχιτεκτονική, τον επεξεργαστή, το λειτουργικό σύστημα κ.α.

Πειραματική Αξιολόγηση

Προκειμένου να ελέγξουμε την λειτουργικότητα και την απόδοση της υλοποίησης μας πραγματοποιήσαμε δύο πειραματικές αξιολογήσεις. Και στα δύο πειράματα γράψαμε σε Noisy

και σε C μια σειρά προγραμμάτων, σχετικά με ενσωματωμένα συστήματα, τα μεταγλωττίσαμε χρησιμοποιώντας τρία διαφορετικά επίπεδα βελτιστοποίησης (-O0, -O1, -O2), και έπειτα τα τρέξαμε και συγκρίναμε την απόδοσή τους σε ότι αφορά το στατικό μέγεθος του τελικού εκτελέσιμου, καθώς και τον χρόνο εκτέλεσης τους (μετρώντας τον αριθμό των εντολών που εκτελέστηκαν). Τα προγράμματα αυτά είναι μία γεννήτρια αριθμών fibonacci (fib), κάποιες ρουτίνες μετατροπής δεδομένων απο αισθητήρες (bme680), δύο βαθυπερατά φίλτρα (lowPassFilter61, lowPassFilter183), ο αλγόριθμος γρήγορης ταξινόμησης (quicksort), ένας μετρητής βημάτων (pedometer), και ένα εκτεταμένο φίλτρο Κάλμαν για ένα εκκρεμές (pendulumEKF). Στο πρώτο πείραμα, μεταγλωττίσαμε και τρέξαμε τα προγράμματα σε έναν επεξεργαστή αρχιτεκτονικής x80_86 για Linux λειτουργικό σύστημα. Στο δεύτερο πείραμα τα προγράμματα τα μεταγλωττίσαμε και τα τρέξαμε στον προσομοιωτή μικροαρχιτεκτονικών Sunflower [8, 9] για 4 διαφορετικές επεκτάσεις (IMFD, IMF, IM, I) της αρχιτεκτονικής RISC-V 32-bit. Στις εικόνες 1.1, 1.2, 1.3 παρουσιάζουμε το πλήθος των δυναμικών εντολών που εκτελούνται, όταν τρέχουμε την πρώτη πειραματική αξιολόγηση. Για το ίδιο πείραμα παρουσιάζουμε το στατικό μέγεθος των προγραμμάτων στην εικόνα 1.4. Στον πίνακα 1.1 συγκρίνουμε το πλήθος των εντολών που εκτελούνται για τα ίδια προγράμματα γραμμένα σε C και Noisy. Στον πίνακα 1.2 παρουσιάζουμε τη σύγκριση του στατικού μεγέθους των προγραμμάτων γραμμένα σε C και Noisy.

Figure 1.1: *Πλήθος δυναμικών εντολών για τα προγράμματα bme680 και fib.*

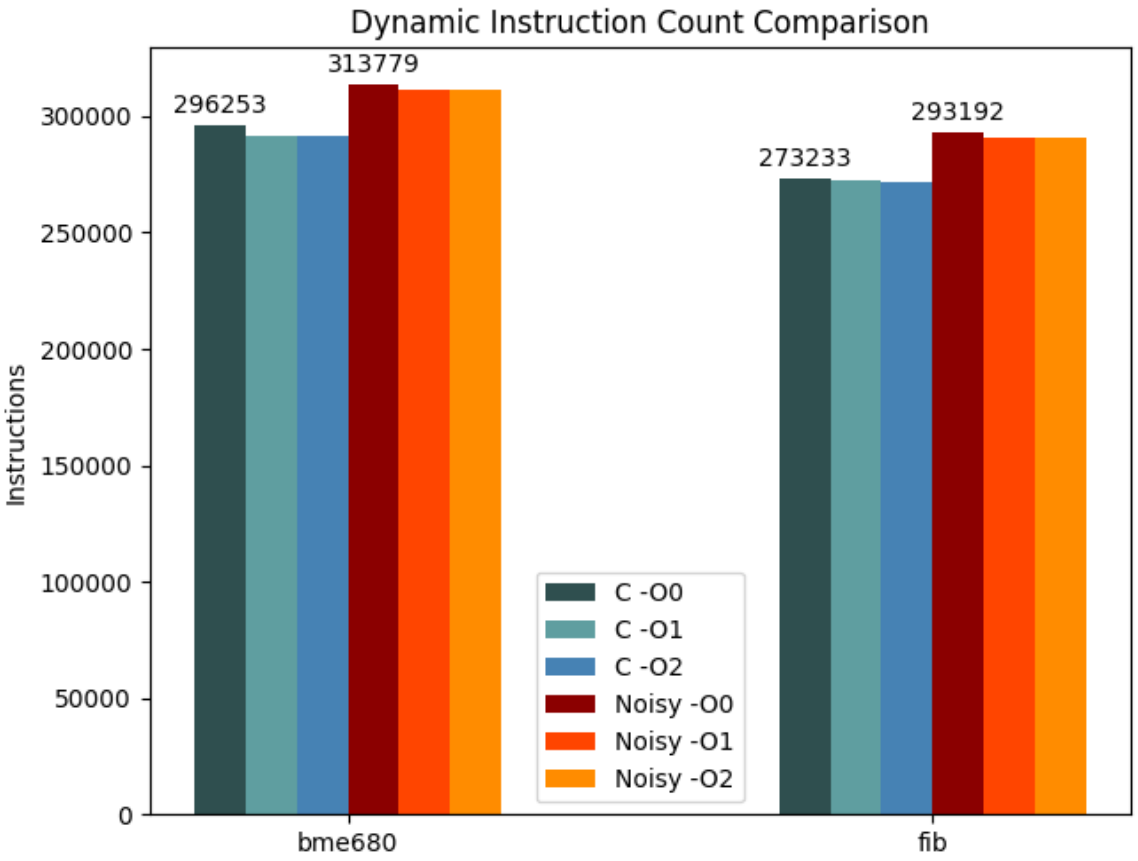


Figure 1.2: *Πλήθος δυναμικών εντολών για τα προγράμματα lowPassFilter183, lowPassFilter61 και quicksort.*

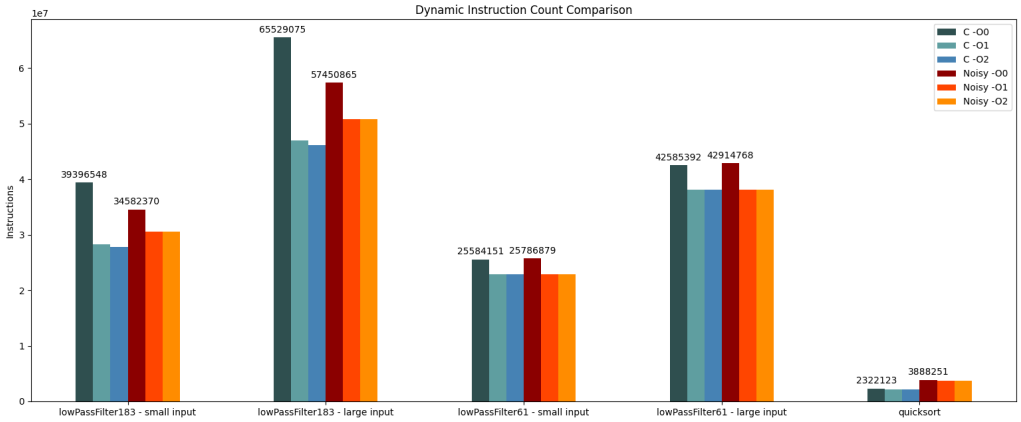


Figure 1.3: *Πλήθος δυναμικών εντολών για τα προγράμματα pedometer και pendulumEKF.*

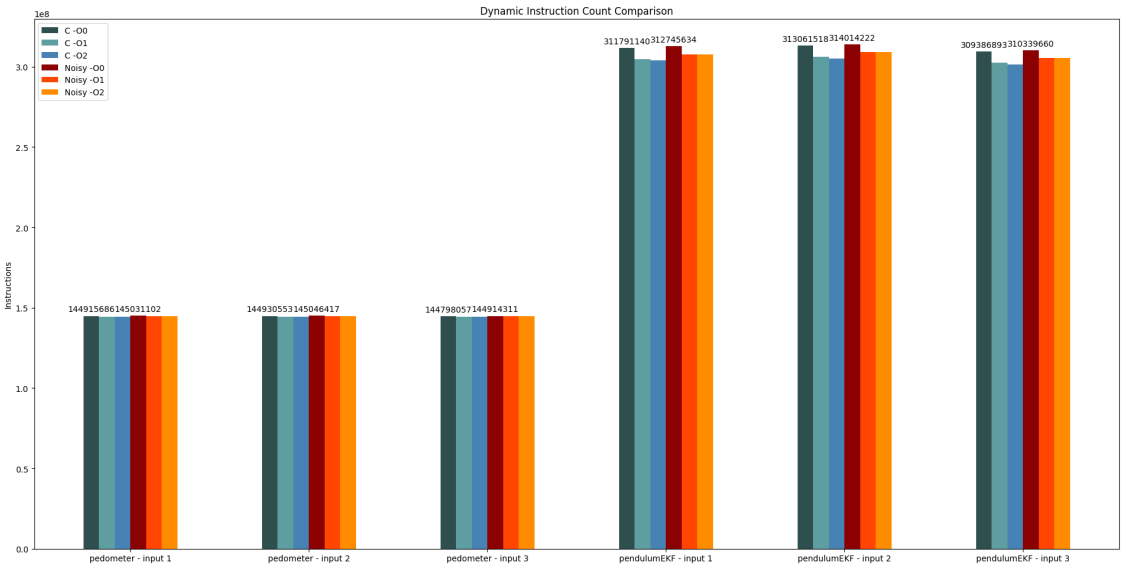
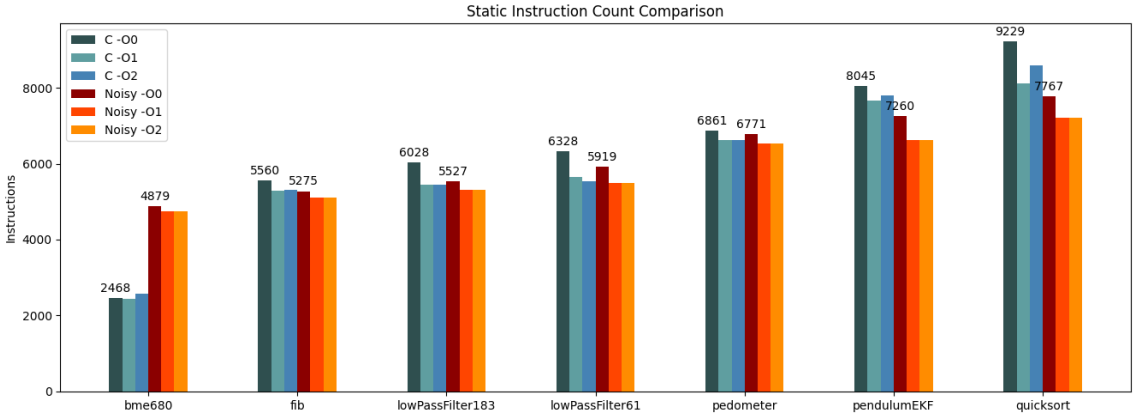


Figure 1.4: *Στατικό μέγεθος όλων των προγραμμάτων.*



Program	**-O0**	**-O1**	**-O2**
bme680	5.92%	6.85%	6.87%
fib	7.30%	6.63%	7.02%
lowPassFilter183 - all inputs	-12.27%	8.06%	10.05%
lowPassFilter61 - all inputs	0.79%	-0.16%	-0.16%
pedometer - all inputs	0.08%	0.20%	0.23%
pendulumEKF - all inputs	0.31%	0.96%	1.27%
quicksort	67.44%	76.41%	76.71%
Mean	11.6%	16.5%	17%
Mean w/o quicksort	0.43%	4.58%	5.06%

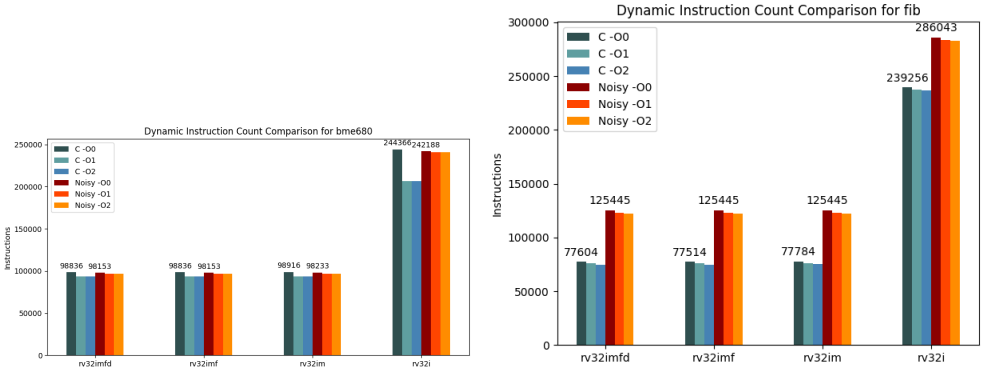
Table 1.1: *Ποσοστιαία διαφορά δυναμικών εντολών που εκτελούνται μεταξύ των υλοποιήσεων σε C και Noisy.*

Program	-O0	-O1	-O2
bme680	97.69%	94.86%	83.38%
fib	-5.13%	-3.42%	-3.85%
lowPassFilter183	-8.31%	-2.51%	-2.51%
lowPassFilter61	-6.46%	-2.78%	-0.81%
pedometer	-1.31%	-1.22%	-1.22%
pendulumEKF	-9.76%	-13.50%	-15.05%
quicksort	-15.84%	-11.07%	-16.04%
Mean	7.27%	8.62%	6.27%
Mean w/o bme680	-7.80%	-5.75%	-6.58%

Table 1.2: *Ποσοστιαία διαφορά του στατικού μεγέθους των προγραμμάτων σε C και Noisy.*

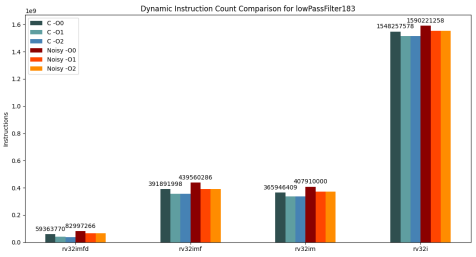
Στη συνέχεια, παρουσιάζουμε τις μετρήσεις μας για την δεύτερη πειραματική αξιολόγηση στον προσομοιωτή Sunflower. Στην εικόνα 1.5 παρουσιάζουμε το πλήθος των εντολών που εκτελούνται για όλα τα προγράμματα και τις διαφορετικές επεκτάσεις της αρχιτεκτονικής RISC-V 32-bit. Στην εικόνα 1.6 παρουσιάζουμε το στατικό μέγεθος των τελικών εκτελέσιμων των προγραμμάτων για την αρχιτεκτονική RISC-V 32-bit. Στους πίνακες 1.3, 1.4 παρουσιάζουμε

την ποσοστιαία διαφορά στις εντολές που εκτελούνται για όλα τα προγράμματα, για όλες τις επεκτάσεις της αρχιτεκτονικής RISC-V 32-bit. Στους πίνακες 1.5, 1.6 συγκρίνουμε το μέγεθος των τελικών εκτελέσιμων για όλα τα προγράμματα σε C και Noisy, για όλες τις επεκτάσεις της αρχιτεκτονικής RISC-V 32-bit.

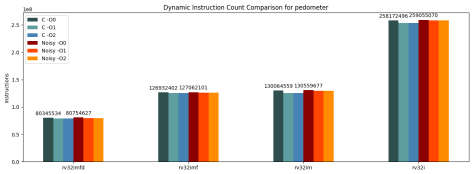


(a) *BME680*

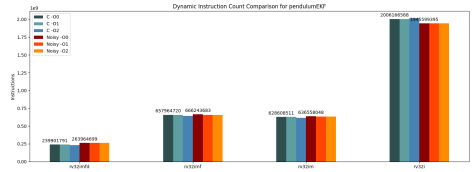
(b) *Fibonacci*



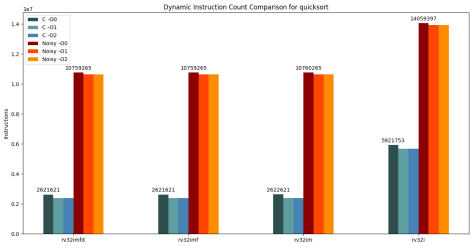
(c) *LowPassFilter183*



(d) *Pedometer*

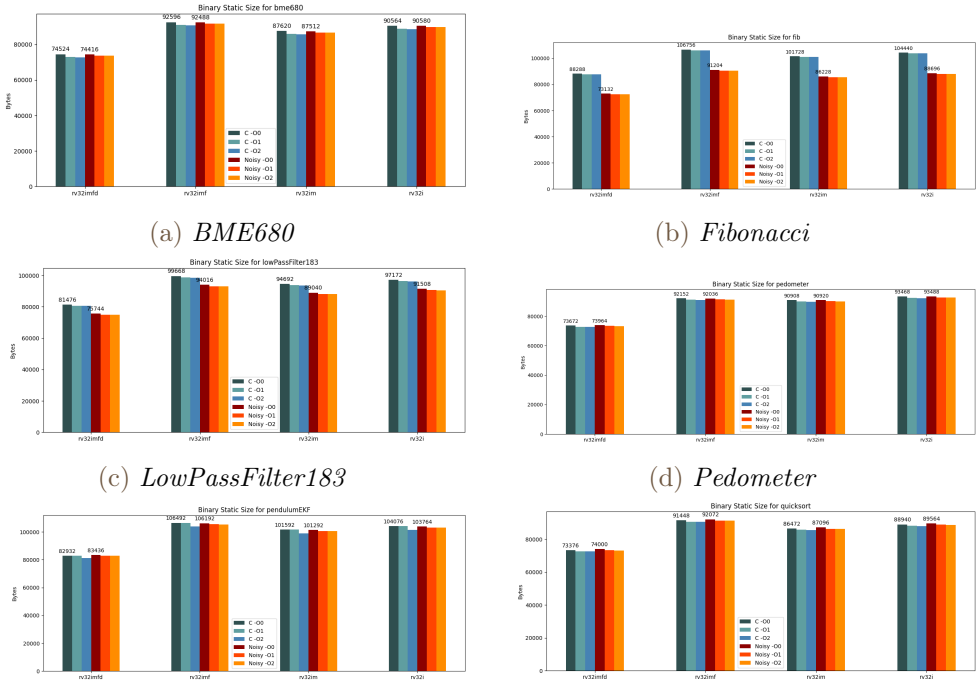


(e) *Pendulum*



(f) *Quicksort*

Figure 1.5: *Πλήθος δυναμικών εντολών για την αρχιτεκτονική RISC-V 32-bit.*



(a) *BME680* (b) *Fibonacci*

(c) *LowPassFilter183* (d) *Pedometer*

(e) *Pendulum* (f) *Quicksort*

Figure 1.6: *Πλήθος στατικών εντολών για την αρχιτεκτονική RISC-V 32-bit.*

RISC-V Extension		IMFD			IMF	
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.69%	3.42%	3.44%	-0.69%	3.42%	3.44%
fib	61.65%	61.74%	63.46%	61.84%	61.94%	63.66%
lowPassFilter183	39.81%	64.01%	76.28%	12.16%	9.46%	9.75%
pedometer	0.51%	1.28%	1.27%	0.10%	0.58%	0.58%
pendulumEKF	10.03%	9.62%	13.58%	1.26%	-0.12%	2.52%
quicksort	310.41%	346.59%	348.34%	310.41%	346.59%	348.34%
Mean	70.29%	81.11%	84.40%	64.18%	70.31%	71.38%
Mean w/o quicksort	22.26%	28.02%	31.61%	14.93%	15.06%	15.99%

Table 1.3: *Ποσοστιαία διαφορά των εντολών που εκτελούνται δυναμικά μεταξύ των υλοποίσεων σε C και Noisy, για τις επεκτάσεις IMFD και IMF της αρχιτεκτονικής RISC-V 32-bit.*

RISC-V Extension		IM			I	
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.69%	3.42%	3.44%	-0.89%	16.63%	16.65%
fib	61.27%	61.36%	63.07%	19.56%	19.28%	19.63%
lowPassFilter183	11.47%	9.99%	10.30%	2.71%	2.47%	2.53%
pedometer	0.38%	3.60%	3.59%	0.34%	1.94%	1.94%
pendulumEKF	1.26%	0.97%	3.03%	-3.02%	-3.11%	-3.70%
quicksort	310.29%	346.45%	348.20%	137.42%	145.24%	145.60%
Mean	64.00%	70.97%	71.94%	26.02%	30.41%	30.44%
Mean w/o quicksort	14.74%	15.87%	16.69%	3.74%	7.44%	7.41%

Table 1.4: *Ποσοστιαία διαφορά των εντολών που εκτελούνται δυναμικά μεταξύ των υλοποίσεων σε C και Noisy, για τις επεκτάσεις IM και I της αρχιτεκτονικής RISC-V 32-bit.*

RISC-V Extension		IMFD			IMF	
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.14%	1.02%	1.12%	-0.12%	0.82%	0.90%
fib	-17.17%	-17.30%	-17.40%	-14.57%	-14.67%	-14.75%
lowPassFilter183	-7.04%	-7.22%	-7.10%	-14.57%	-14.67%	-14.75%
pedometer	0.40%	0.74%	0.86%	-0.13%	0.11%	0.23%
pendulumEKF	0.61%	-0.07%	2.06%	-0.28%	-0.94%	1.39%
quicksort	0.85%	0.91%	0.95%	0.68%	0.73%	0.76%
Mean	-3.75%	-3.65%	-3.25%	-3.35%	-3.29%	-2.86%

Table 1.5: *Ποσοστιαία διαφορά του στατικού μεγέθους των προγραμμάτων σε C και Noisy, για τις επεκτάσεις IMFD και IMF της αρχιτεκτονικής RISC-V 32-bit.*

RISC-V Extension		IM			I	
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.12%	0.86%	0.95%	0.02%	1.08%	1.14%
fib	-15.24%	-15.35%	-15.43%	-15.07%	-15.18%	-15.27%
lowPassFilter183	-5.97%	-6.11%	-6.01%	-5.83%	-5.97%	-5.87%
pedometer	0.01%	0.32%	0.40%	0.02%	0.36%	0.44%
pendulumEKF	-0.30%	-0.98%	1.46%	-0.30%	-0.97%	1.41%
quicksort	0.72%	0.78%	0.80%	0.70%	0.75%	0.78%
Mean	-3.48%	-3.41%	-2.97%	-3.41%	-3.32%	-2.89%

Table 1.6: *Ποσοστιαία διαφορά του στατικού μεγέθους των προγραμμάτων σε C και Noisy, για τις επεκτάσεις IM και I της αρχιτεκτονικής RISC-V 32-bit.*

Από την πειραματική αξιολόγηση προκύπτει ότι οι υλοποιήσεις των προγραμμάτων σε Noisy, όσον αφορά τον δυναμικό χρόνο εκτέλεσης, έχουν λίγο χειρότερη απόδοση σε σχέση με τις υλοποιήσεις σε C, και συγκεκριμένα εκτελούν κατά μέσο όρο 5.06% περισσότερες εντολές, στο πείραμα σε Linux για σημαία βελτιστοποίησης -O2. Στο πείραμα για την αρχιτεκτονικη RISC-V τα προγράμματα σε Noisy, εκτελούν κατά μέσο όρο 17,93% περισσότερες εντολές για σημαία βελτιστοποίησης -O2, για όλες τις διαφορετικές επεκτάσεις της αρχιτεκτονικής.

Στους μέσους όρους αυτούς, δεν έχουμε συμπεριλάβει την υλοποίηση του προγράμματος quicksort, η οποία έχει πολύ χειρότερη απόδοση σε σχέση με τα υπόλοιπα προγράμματα. Αυτή η διαφορά της απόδοσης οφείλεται στον μηχανισμό περάσματος παραμέτρων πινάκων από τη γλώσσα, ο οποίος αντιγράφει τις τιμές των πινάκων αντί να τους περνάει κατ΄ αναφορά και δημιουργεί παραπάνω καθυστερήσεις. Το δεύτερο συμπέρασμα που προκύπτει από την πειραματική αξιολόγηση είναι ότι και στις δύο πειραματικές διατάξεις τα προγράμματα γραμμένα σε Noisy έχουν μικρότερο στατικό μέγεθος σε σχέση με τα αντίστοιχα προγράμματα σε C. Συγκεκριμένα, στο πρώτο πείραμα τα προγράμματα σε Noisy έχουν κατά μέσο όρο 6.58% μικρότερο στατικό μέγεθος για σημαία βελτιστοποίησης -O2. Σε αυτόν τον μέσο όρο, δεν έχουμε υπολογίσει το πρόγραμμα bme680, καθώς έχει πολύ μεγαλύτερο στατικό μέγεθος, γεγονός το οποίο οφείλεται στο ότι στην παρούσα υλοποίηση της γλώσσας δεν υπάρχουν δομές δεδομένων structs. Για το δεύτερο πείραμα παρατηρούμε ότι τα προγράμματα σε Noisy έχουν περίπου 3% μικρότερο στατικό μέγεθος σε όλες τις επεκτάσεις της αρχιτεκτονικής RISC-V 32-bit.

Μελλοντική Εργασία

Στην παρούσα εργασία υλοποιήσαμε το κύριο υπολογιστικό μέρος της γλώσσας, τους στατικούς πίνακες και μέρος του συστήματος που αφορά το μοντέλο των Επικοινωνουσών Ακολουθιακών Διεργασιών. Παρ΄ όλα αυτά, υπάρχουν ακόμα μέρη της γλώσσας που μένουν να υλοποιηθούν σε μελλοντική εργασία. Αρχικά, το σύστημα τύπων της γλώσσας χρειάζεται να εμπλουτιστεί για νέες δομές δεδομένων, καθώς και να συμπεριλάβει τους τύπους που αφορούν διαστάσεις και φυσικά μεγέθη των μεταβλητών. Στην ίδια εργασία θα μπορούσε να μελετηθεί και η διασύνδεση του συστήματος τύπων της Noisy με την συμπληρωματική της γλώσσα τη Newton. Το δεύτερο ζήτημα που θα μπορούσε να μελετηθεί σε μελλοντική εργασία, είναι η δημιουργία ενός συστήματος χρόνου εκτέλεσης που να διαχειρίζεται τις Επικοινωνούσες Ακολουθιακές Διεργασίες, και τη μνήμη που αφορά δυναμικές δομές δεδομένων. Ένα ακόμη κομμάτι της γλώσσας, είναι ό,τι αφορά την διαχείριση της αβεβαιότητας και των σφαλμάτων στις τιμές διαφόρων μεταβλητών που είναι ένα ακόμα ιδιαίτερο χαρακτηριστικό της Noisy. Τέλος, η γλώσσα παρέχει και ένα Δηλωτικό υποσύνολο, το οποίο έχει ειδικό συντακτικό και σημασιολογία και βοηθάει στη δήλωση ιδιοτητών και αναλοίωτων συνθηκών που πρέπει να ισχύουν σε ένα πρόγραμμα. Το δηλωτικό υποσύνολο, επιτρέπει στον προγραμματιστή να αποδείξει ή να διαψεύσει ιδιότητες που αφορούν τα προγράμματα του, και επομένως να δημιουργήσει κώδικα πιο ασφαλή και με λιγότερα λάθη. Όλες οι παραπάνω προσθήκες στη γλώσσα, μπορούν να πατήσουν πάνω στη δικιά μας υλοποίηση και να χρησιμοποιήσουν τα εργαλεία του LLVM για να υλοποιήσουν πολλές από τις ζητούμενες λειτουργικότητες.

Chapter 2

Introduction

Nowadays we are becoming more and more reliant to microcomputers and small computing devices that interact with their environment. The computers in cars, planes, smartphones all have this in common. They receive data from various sensors and process them in real-time in order to provide their users with important information for critical or not applications. All these computing devices need to offer low energy consumption and good power efficiency since their resources are limited. Also, since these devices need to operate in real-time this means that these systems have to guarantee a response within specific time constraints. Therefore, a reasonably good computation performance is required. All these devices are called embedded systems. More specifically, an embedded system is a computer system —a combination of a computer processor, computer memory, and input/output peripheral devices— that has a dedicated function within a larger mechanical or electronic system [1, 2].

Since embedded systems interact with the physical world in the form of sensor readings it becomes apparent that measurement uncertainty plays an important role in the success of those systems. Measurement uncertainty is the expression of the statistical dispersion of the values attributed to a measured quantity [10]. This uncertainty can stem from noise in the measuring instruments, data losses or alterations (e.g. bit flips), transformations from analog to digital values etc. Since it plays an important role in the correct functionality of systems, the uncertainty must be taken into account when designing or writing code for these devices. Also, since these platforms measure and compute values for real physical systems, many of their computations are governed by physics rules and invariants. For example a calculated temperature (in Kelvin) cannot have a negative value or a plane cannot fly above a very high altitude. These invariants and physical rules should also be taken into consideration as a means of detecting, avoiding and fixing errors of programs either when designing an application or even when the application is running. In order to solve this problem and also in order to meet the specified requirements of the applications many solutions have emerged and are used, making breakthrough-progress every day. These solutions range from better processors and sensors to new algorithms, better implementations of old algorithms, compiler optimizations and many other improvements. These improvements facilitate designers of embedded systems as well as improve our every day appliances with more effective devices and better programs. However, all these tools require a lot of engineering effort and each system needs its own fine tuning and setup in

order to work efficiently. Therefore, a need for better and more specialized tools is created. These tools need to model, measure and reduce uncertainty, simplify the interconnection of hardware (processors, sensors, memory) with software, while maintaining a standard in performance and energy efficiency.

Despite the ever changing needs of the embedded systems and the real world, the C programming language ecosystem remains a constant for many years. C is one of the oldest, most stable and supported language in the world. It compiles to extremely compact and efficient machine code and it makes the perfect candidate for its job. Nevertheless, C is an old language and does not directly support many newer programming constructs and idioms, let alone specific constructs for real-time applications (e.g. uncertainty measurement constructs). Modifying C and its compiler infrastructure has been a difficult task to do and with the need for new and more specialized tools for embedded systems, the need for a new programming model also has emerged.

Aims of this Thesis

Noisy is a newborn programming language that aims to facilitate embedded systems programmers with constructs designed for the real world and aspires to meet the growing need for new tools. Noisy is designed by professor Phillip Stanley-Marbell and in this thesis we seek to study and present the main features of this new language as well as develop an implementation of its bare minimum functionalities. Programs written in Noisy can be compiled and run targeting a plethora of machines and architectures all thanks to LLVM compiler infrastructure. Since the Noisy language reference contained many ideas and proposals for functionalities and language constructs, it was not possible to implement all of them, but only a small part of the language (the basic computational core). As a result, a lot of interesting work needs to be done in the future. Finally, this thesis aims to act as documentation for Noisy, its features and their implementation.

Chapter 3

Theoretical Background

Before diving further in the Noisy design and implementation it is important to understand the basic concepts which will encounter in the next Chapters. Noisy is a programming language meaning that is a notation that describes calculations to both humans and computers. Programming languages are really important in the modern world because most of the software that is currently running on all computers has been written in a programming language. However, in order for the computer to be able to understand and execute a program written in a programming language, a specific kind of program that translates programming languages to machine code is needed. In general, those programs that receive as input a program written in a programming language (source language) and produce as output a program written in another language (target language) are called compilers. Therefore in this Section we will examine tha basic theoretical background for compiler design and implementation [11]. Also, as we will see in Chapter 4, Noisy aims by design to implement the Communicating Sequential Processes (CSP) model. In this chapter we will also explain its basics.

3.1 Compiler Overview

As we have seen compilers are programs that translate programs from human readable form to an executable form. Compilers, apart from translating programs, also play an important role in finding and reporting errors in the source program. Nowadays, compilers are an irreplaceable part of all modern programming languages and therefore their study and development is really important and beneficial.

3.1.1 Structure of a compiler

Compilers are perceived as a black box that takes as input a source program and generates a semantically equal target program. By trying to understand how this process works it becomes apparent that two separate tasks are needed. The first one is the analysis of the source program and the second one is the synthesis of the target program. The analysis of the source program is commonly known as the front-end of the compiler while the synthesis of the target is known as the backend. Both frontend and backend consist of different distinct stages each of which uses the output of the previous stage as input. The frontend of the compiler comprises of the lexical analysis, syntactic analysis and semantic analy-

sis. The backend of the compiler usually has the code generation stage, code optimization stage and finally linking stage. The compiler uses an Intermediate Representation (IR) of the program so the different stages of the analysis can work together. This Intermediate Representation is the output of the frontend of the compiler and also acts as the input of the backend of the compiler. The most used IR is a tree structure called the Abstract Syntax Tree (AST). The AST is semantically equal with the source program and it is the final form before code generation can begin.

3.1.2 Lexical analysis

The first stage of a compiler is the lexical analysis. Lexical analysis reads the input stream of characters of the source program, groups them into sequences with semantic value called lexemes. For each lexeme, the lexical analyzer produces a lexical token which is stored and used by the next stage of the compiler, the syntactic analysis. These lexical tokens are simply the words that make up the source program. Since lexical analyzer is the part of the compiler that directly reads the source program it also does other useful tasks. For example, it eliminates whitespace characters (space, tabs, newline characters), ignores comments and it keeps information about the source code current line and characters that is useful for error message printing. Each programming language has a specific set of words that are valid for the language, which is called the vocabulary of the language. The output of lexical analysis is the sequence of the accepted tokens.

3.1.3 Syntactic analysis

The second stage of the frontend is the syntactic analysis or parsing. Each language during its design stage has a specific set of syntactic rules that define the structure of its well formed programs. Every program that does not conform to those rules of the language is not considered a correct program, meaning it cannot compile. Syntactic rules of a language can be expressed in the form of context-free grammars or in BNF notation (Backus-Naur Form). Depending on the form and the characteristics of a language's grammar the techniques for syntactic analysis may differ. However, all syntactic analyzers (parsers) take as input the token sequence created by the lexical analysis and return as output a semantically equal representation of the program in a tree-structure called the Abstract Syntax Tree (AST) which is also the input for the next stage of the compiler, the semantic analysis. In some cases semantic analysis can be done during the syntactic analysis.

Context Free Grammars

A context-free grammar G is defined by the 4-tuple $G = (V, \Sigma, R, S)$, where V is a finite set; each element $v \in V$ is called a non-terminal character or a variable [12]. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G. Σ is a finite set of terminals, disjoint from V, which make up the actual content of

the sentence. The set of terminals is the alphabet of the language defined by the grammar G. R is a finite relation in $V \times (V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of R are called the (rewrite) rules or productions of the grammar (also commonly symbolized by a P). S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

A production rule in R is formalized mathematically as a pair $(\alpha, \beta) \in R$, where $\alpha \in V$ is a non-terminal and $\beta \in (V \cup \Sigma)^*$ is a string of variables and/or terminals; rather than using ordered pair notation, production rules are usually written using an arrow operator with α as its left hand side and β as its right hand side: $\alpha \to \beta$. It is allowed for β to be the empty string, and in this case it is customary to denote it by ϵ. The form $\alpha \to \varepsilon$ is called an ϵ-production [13]. It is common to list all right-hand sides for the same left-hand side on the same line, using | (the pipe symbol) to separate them. Rules $\alpha \to \beta_1$ and $\alpha \to \beta_2$ can hence be written as $\alpha \to \beta_1 \mid \beta_2$. In this case, β_1 and β_2 are called the first and second alternative, respectively.

For any strings $u, v \in (V \cup \Sigma)^*$, u directly yields v, written as $u \Rightarrow v$ if $\exists(\alpha, \beta) \in R$ with $\alpha \in V$ and $u_1, u_2 \in (V \cup \Sigma)^*$ such that $u = u_1 \alpha u_2$ and $v = u_1 \beta u_2$. Thus, v is a result of applying the rule (α, β) to u.

For any strings $u, v \in (V \cup \Sigma)^*$, u yields v or v is derived from u if there is a positive integer k and strings $u_1, \cdots, u_k \in (V \cup \Sigma)^*$ such that $u = u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k = v$. This relation is denoted $u \overset{*}{\Rightarrow} v$, or $u \Rightarrow v$ in some textbooks. If $k \geq 2$, the relation $u \overset{+}{\Rightarrow} v$ holds. In other words, $(\overset{*}{\Rightarrow})$ and $(\overset{+}{\Rightarrow})$ are the reflexive transitive closure (allowing a string to yield itself) and the transitive closure (requiring at least one step) of $(\Rightarrow)$, respectively.

The language of a grammar $G = (V, \Sigma, R, S)$ is the set $L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$ of all terminal-symbol strings derivable from the start symbol. A language L is said to be a context-free language (CFL), if there exists a CFG G, such that $L = L(G)$.

Parsing techniques

Parsing is the process of determining how a symbol series can be produced by a grammar, or more specifically how the input can be derived from the start symbol of the grammar. There are essentially two ways to do this.

- **Top-down parsing:** This technique is the attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right.

- **Bottom-up parsing:** This technique starts with the input and attempts to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on.

In general top-down parsing and bottom-up parsing can be a difficult task meaning it can require complexity up to $O(n^3)$. However, in cases of specific grammars it can be simpler to implement achieving a complexity of $O(n)$. Such is the case of LL(1) grammars. LL grammars are the Context Free Grammars that can be parsed by an LL parser. LL parsers are a specific type of top-down parsers that parse the input from **L**eft to right and

construct a **L**eftmost derivation of the sentence (hence LL). In general, LL(k) grammars are defined as:
$G = (V, \Sigma, R, S)$ is an LL(k) grammar if, for arbitrary derivations
$S \Rightarrow^L w_1 A\chi \Rightarrow w_1\nu\chi \Rightarrow^* w_1 w_2 w_3$
$S \Rightarrow^L w_1 A\chi \Rightarrow w_1\omega\chi \Rightarrow^* w_1 w_2' w_3',$
when the first k symbols of $w_2 w_3$ agree with those of $w_2' w_3'$, then $\nu = \omega$ [14].

Informally, when a parser has derived $w_1 A w_3$, with A its leftmost non-terminal and w_1 already consumed from the input, then by looking at that w_1 and peeking at the next k symbols w of the current input, the parser can identify with certainty the production rule r for A. The symbols peeked are also called lookahead tokens. Therefore an LL(1) grammar is an LL grammar that needs only 1 lookahead token (the current input) in order to identify the productions of the grammar. LL(1) parsers are important because they are effective and easy to implement by hand.

FIRST and FOLLOW sets

FIRST and FOLLOW sets are functions that help to create top-down or bottom-up parsers for a given grammar G. These sets allow us to decide which production rule will be applied, based on the next input token. We define $FIRST(\alpha)$ to be the set of terminals that start strings that are produced by α, where α can be any string of grammar symbols. If $\alpha \stackrel{\star}{\rightarrow} \epsilon$, where ϵ is the empty string symbol, then ϵ also belongs to the FIRST(α). We define FOLLOW(A) for non-terminal A, to be the set of terminals α that can appear immediately to the right side of A in a sentential form. If A is the rightmost symbol in a sentential form, then "$" symbol belongs to FOLLOW(A), where "$" symbol is the special end symbol that does not belong to any grammar.

In order to calculate FIRST(X) for all symbols of a grammar X, we apply the following rules until we cannot add another terminal or ϵ to any FIRST set.

1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$, then place α to FIRST(X), if for some i, *alpha* belongs to FIRST(Y_i) and ϵ belongs to all $FIRST(Y_1), \dots, FIRST(Y_{i-1})$.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

In order to calculate FOLLOW(A) for all non-terminal symbols A, we apply the following rules until we cannot add anything else to any FOLLOW set.

1. Place "$" symbol to FOLLOW(S), where S is the starting symbol of the grammar and "$" symbol is the rightmost end of the input.
2. If a production $A \rightarrow \alpha B \beta$ exists, then all elements of FIRST(B) belong to FOLLOW(B) apart from ϵ.
3. If a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ exists, where FIRST(β) contains ϵ, then all elements of FOLLOW(A) belong to FOLLOW(B).

Parsing LL(1) grammars

In order to parse an LL(1) grammar, we use the information of FIRST and FOLLOW sets so we can create a predictive parsing table $M[A, \alpha]$, which is a two-dimensional array, where A is a non-terminal symbol and α is a terminal or "$" symbol. The parser can then use this table in order to parse the input in order to decide which production is selected for each input token. The selection is based on the following idea; a production $A \to \alpha$ is selected if the next input token belongs to FIRST(α). When $\alpha \xrightarrow{\star} \epsilon$, the production $A \to \alpha$ is selected when the current input token belongs to FOLLOW(A). If the input token is the "$" symbol, then $A \to \alpha$ is selected when "$" belongs to FOLLOW(A). In order to create the predictive parsing table M, we use the following algorithm:
INPUT: Grammar G.
OUTPUT: Predictive parsing table M.
METHOD: For every production $A \to \alpha$ of the grammar G, do:

1. For every terminal α that belongs to FIRST(α), add $A \to \alpha$ production to $M[A, \alpha]$
2. If ϵ belongs to FIRST(α), then for every terminal b that belongs to FOLLOW(A), add $A \to \alpha$ production to $M[A, b]$. If ϵ belongs to FIRST(α) and $ belongs to FOLLOW(A), also add $A \to \alpha$ to $M[A, \$]$.

If there is no production in $M[A, \alpha]$, after the completion of the algorithm, then we consider $M[A, \alpha]$ to be an error, which is represented as a blank entry in the table.

This algorithm can be used for any grammar G in order to produce an M table. For every LL(1) grammar, each entry in the table represents uniquely a production rule or a syntactic error. If an M table has duplicate entries then it means that grammar G is not LL(1), although for some grammars we can use transformations in order to convert them to LL(1) grammars. During compilation, the parser uses the information of the M table for the given grammar G of a language L, and can determine if a program, written in L, is syntactically correct and can then create the AST of the given program in linear time.

3.2 Communicating Sequential Processes model

The Communicating Sequential Processes is a formal language for describing concurrent systems whose component processes interact with each other by communication [15]. The term originates from Tony Hoare's original article in 1978 [3], in which it was described as a concurrent programming language. In the original paper, programs in CSP were written as a parallel composition of sequential processes communicating with each other through synchronous message-passing. Later, the term was evolved to be a mathematical theory, member of the family of mathematical theories known as process algebras, by adding mathematically defined semantics [16] and by adding representation for unbounded nondeterminism [17]. The CSP model allows us to describe complex systems using primitive elements of processes and algebraic process operators that denote the different ways that processes can communicate. Processes execute independently and all communication is based on synchronous message-passing through named channels and is considered to

happen instantaneously. Communication can happen only when all its participants are ready to perform it (handshake communication) and we use the term *event* to refer to communication actions. CSP also offers notation to express bad forms of behavior such as as deadlocks, divergence and nondeterminism, and thus it enables us to reason about them and hopefully avoid them in a practical application.

3.2.1 Primitives

Events

Events are the most basic objects in CSP and refer to communication events between processes. We can think of events not only as the one-way transmission of data but also as a transaction or synchronization between two or more processes. Processes' only way of interacting with the environment is through events and events are the only thing that can be observed by the environment. Also, events are considered to happen instantaneously and all participants of a communication need to be ready to perform it, in order for an event to execute. In the next Sections we will use lowercase english letters to refer to events.

Processes

Processes are sequential commands that can communicate with their environment or other processes exclusively by communication events. We use capital letters to represent them. Also there are two primitives processes defined in CSP the *SKIP* and *STOP* processes. STOP process does nothing and never communicates and it can be thought as a deadlock. SKIP, on the other hand does also nothing but terminates its execution successfully and thus represents the successful termination.

3.2.2 Operators

Operators are used on events and processes to represent the various necessary concepts of the ways that processes can communicate and create more complex programs that can describe interactions of the real world. The most used operators are presented below:

Prefix

Given an event a and a process P, $a \rightarrow P$, is the process that communicates a with the environment and then acts as P. Using *STOP* and prefixing, together, allows us to create a finite sequence of steps before stopping.

Deterministic choice

Given the events a,b and processes P,Q, $(a \rightarrow P) \Box (b \rightarrow Q)$ behaves like process P if the environment communicates event a or behaves like process Q if environment communicates event b. Therefore, this operator allows the future evolution of a process to be defined a choice between two processes and the environment resolves the choice by communication.

Nondeterministic choice

Nondeterministic choice is similar to the deterministic choice in the sense that allows the future evolution of the process to be a choice between two component processes, but it is different because it does not allow the environment to choose between processes by communication. Thus, for events a, b and processes P, Q, $(a \to P) \sqcap (b \to Q)$ can behave either like $(a \to P)$ or like $(b \to Q)$ but it is obliged to communicate either a or b only if the environment offers both a and b. The environment cannot affect which will be chosen by communicating either a or b and in a case where only one of a or b is present the process can reject either.

Interleaving

The Interleaving operator represents that processes run completely independently of each other. Given two processes P, Q, the process $P|||Q$ executes as both P and Q simultaneously and the events from both processes are interleaved in time in a nondeterministic manner.

Interface parallel

The interface parallel operator represents processes that execute in parallel but require synchronization on a specific event. An event in the interface set can occur only when all component processes are ready to execute it. Given processes P, Q and an event a, the process $P|[\{a\}|]Q$ acts as P and Q but they both must be able to perform a.

Hiding

The hiding operator represents events that are invisible to and uncontrollable by the environment. Given a process P and an event a, the process $P \backslash \{a\}$ behaves like P, but the event a is hidden.

Other operators

There are also other operators that define different interactions such as piping, enslavement, alphabetized parallel composition and indexed choices.

3.2.3 Example

Using the CSP language we can describe and model behavior for various systems and their interactions. We will use a toll station machine as our example and its interactions with a vehicle willing to pass through the toll station. The toll station is able to carry out two different events, "cash" and "pass", which represent the payment and the opening of the bar that allows the vehicle to pass. A toll station machine that demands payment only in cash can be written as:
$TollMachine = cash \to pass \to STOP$.
A vehicle whose driver wants to pass the toll station and pay either by card or cash, can

be modelled as:
$Vehicle = (cash \rightarrow STOP) \square (card \rightarrow STOP)$
In this example we see the use of the deterministic choice operator. The two processes of the machine and the vehicle can be put in parallel, so they interact with each other. The behaviour of the composite process depends on the events the two processes must synchronise on. We model the composite process as:
$TollMachine|[\{cash, card\}]|Vehicle \equiv cash \rightarrow pass \rightarrow STOP$
whereas if synchronization was required only on cash payment we would get:
$TollMachine|[\{cash\}]|Vehicle \equiv (cash \rightarrow pass \rightarrow STOP) \square (card \rightarrow STOP)$.
If we wanted to abstract from the process and see it as an external observer, we would use the hiding operator:
$((cash \rightarrow pass \rightarrow STOP) \square (card \rightarrow STOP)) \setminus \{coin, card\} \equiv (pass \rightarrow STOP) \sqcap STOP$.
This process, either lets the vehicle pass and stops or just stops. We see, how we can introduce nondeterminism by examining the same problem from an external point of view.

Now, using the same description we described theoretically, we could implement the same functionalities in a program written in a language supporting this idiom.

3.2.4 Benefits

Implementing the CSP model in a programming language comes with great benefits. The CSP model is a descriptive tool to model and describe many systems and their behavior. Especially, in the domain of embedded systems where we have a lot of devices which communicate with one another, using the CSP model seems very intuitive. Also, the synchronous unbuffered channels as a means of communication, are safer than other communication primitives like semaphores or locks and also are easier to use, because the use of common memory is avoided in favor of channels. Moreover, even though concurrency does not necessarily mean parallelism, parallelism in most cases can easily be achieved with a transformation of the original program, depending on the underlying implementation as well. Finally, the CSP model is a theoretical tool that allows us to reason about program semantics and unwanted program behavior, prove properties of programs and rule out bugs.

Chapter 4

Noisy Language

Sensor driven hardware and embedded platforms require relevant programming abstractions. These abstractions expressed in the form of native language constructs allow compilers to reason about the semantics of programs of the specific domain and thereby implement transformations that improve safety, efficiency or verify correctness. Noisy is a language designed specifically for the domain of sensor-driven hardware platforms. It features primitives for physical signals, units of measure. primitives for accessing measurement uncertainty, primitives for sampling and quantization, operations on timeseries and discrete time sampled data and operations for signal measurements through a CSP like channel interface. Also, Newton [5], a second language complements this facilities but is intended to be used by hardware platform designers. Newton allows hardware designers to describe physical properties of the hardware platform, including sensors, signals that sensors generate, and constraints between those signals dictated by both physical laws and the architecture of the device. These two languages aim to bridge the gap between software and hardware design and interoperate by using a common compiler infrastructure, allowing transformations using information generated both by program semantics and by hardware specifications and constraints.

4.1 Noisy language

Noisy is a language for processing signals from the physical world. Although the capabilities we described earlier could be implemented by libraries for existing languages like C, we believe that it would bear more scientific interest to create a new programming language in order to investigate how compilers can receive and analyze semantic information about physical signals and their interactions and make program transformations that improve safety, performance and efficiency. This semantic information would not be available at compile-time using a library.

4.1.1 Features

Noisy is an imperative, strongly typed, statically checked compiled language that provides features designed for the embedded system and real time application domain. Noisy also aims to implement the CSP model. Furthermore, Noisy's syntactic constructs are influenced heavily by Dijkstra's guarded command language [4].

4.1.2 Syntactic elements

A sequence of Unicode characters, which corresponds to a sequence of one or more tokens described in the vocabulary of the language is a valid program if it conforms to the rules of the language grammar. The language grammar is presented in Listing 4.1 using EBNF notation.

Listing 4.1: Formal grammar of the Noisy language.

```
/*
*       This is the subset of Noisy for which we will create the LLVM backend.
*       This subset exludes the declarative part of Noisy (problem definitions
*       and predicates). Regarding the imperative part, it will support only
*       integer and float arithmetics as well as the most basic data structures.
*/

/*
*   Lexical elements
*/
character       ::= Unicode-0000h-to-Unicode-FFFFh .
reservedOperatorOrSeparatorToken ::=     "~" | "!" | "%" | "^" | "&" | "*" | "(" |
                                         ")" | "," | "-" | "+" | "=" | "/" | ">" |
                                         "<" | ";" | ":" | "'" | "\"" | "{" | "}" |
                                         "[" | "]" | "|" | "<-" | "." | "<=" | ">=" |
                                         "^=" | "|=" | "&=" | "%=" | "/=" | "*=" | "-=" |
                                         "+=" | ":=" | "!=" | ">>" | ">>=" | "<<" |
                                         "<<=" | "<-=" | "&&" | "||"| "::" | "=>" | "++" |
                                         "--" .

reservedIdentifiers     ::=     "and" | "bool" | "complex" | "const" | "false" |
                                "fixed" | "float128" | "float16" | "float32" | "float4" |
                                "float64" | "float8" | "function" | "head" | "include" |
                                "int128" | "int16" | "int32" | "int4" | "int64" |
                                "int8" | "iterate" | "length" | "list" | "load" |
                                "match" | "matchseq" | "module" | "nat128" | "nat16" |
                                "nat32" | "nat4" | "nat64" | "nat8" | "nil" | "of" |
                                "rational" | "return" | "reverse" | "sequence" | "set" |
                                "string" | "tail" | "true" | "type" | "typeannote" |
                                "typemax" | "typemin" | "vector" .

zeroToNine              =   "0-9" .
oneToNine               =   "1-9" .
radix                   =   oneToNine {zeroToNine} "r" .
charConst               =   "'" character "'" .
integerConst            ::= ["+" | "-"] [radix] ("0" | oneToNine {zeroToNine}) |
                                charConst .
boolConst               ::= "true" | "false" .
drealConst              =   ("0" | oneToNine {zeroToNine}) "." {zeroToNine} .
erealConst              =   (drealConst | integerConst) ("e" | "E") integerConst .
realConst               ::= ["+" | "-"] (drealConst | erealConst) .
stringConst             ::= "\"" {character} "\"" .
idchar                  =   char-except-rsvopseptoken .
identifier              ::= (idchar-except-zeroToNine) {idchar} .

/*
*   Syntactic elements: top-level program structure
*/
program         ::= moduleDecl {(moduleDecl | functionDefn)} .
functionDefn    ::= identifier ":" "function" signature "->" signature "=" scopedStatementList.
signature       ::= "(" ((identifier ":" typeExpr {"," identifier ":" typeExpr}) | "nil") ")" .
moduleDecl      ::= identifier ":" "module" "(" typeParameterList ")" "{" moduleDeclBody "}" .
moduleDeclBody  ::= {moduleTypeNameDecl ";"} .

/*
*   Types
*/
moduleTypeNameDecl      ::= identifier ":" (constantDecl | typeDecl | functionDecl) .
constantDecl            ::= "const" (integerConst | realConst | boolConst) .
typeDecl                ::= ("type" typeExpr) .
functionDecl            ::= "function" writeTypeSignature "->" readTypeSignature .
readTypeSignature       ::= signature .
writeTypeSignature      ::= signature .
identifierOrNil         ::= qualifiedIdentifier | "nil" .
identifierOrNilList     ::= identifierOrNil {"," identifierOrNil} .
identifierList          ::= identifier {"," identifier} .
typeExpr                ::= basicType | anonAggregateType | typeName .
typeName                ::= identifier ["->" identifier] .
basicType               ::= "bool" | integerType | realType | "string" .
```

```
integerType             ::= "nat4" | "nat8" | "nat16" | "nat32" | "nat64" | "nat128" | "int4" |
                            "int8" | "int16" | "int32" | "int64" | "int128" .
realType                ::= "float4" | "float8" | "float16" | "float32" | "float64" | "float128".
numericType             ::= integerType | realType .
anonAggregateType       ::= arrayType .
arrayType               ::= "array" "[" integerConst "]" {"[" integerConst "]"} "of" typeExpr.
numericConst            ::= integerConst | realConst .
initList                ::= "{" expression {"," expression} "}" .
idxInitList             ::= "{" element {"," element} "}" .
starInitList            ::= "{" element {"," element} ["," "*" "=>" expression] "}" .
element                 ::= expression [ "=>" expression ] .
typeParameterList       ::= [identifier ":" "type" {"," identifier ":"  "type"}] .

/*
*   Statements
*/
scopedStatementList     ::= "{" statementList "}" .
statementList           ::= {statement} .
statement               ::= [ assignmentStatement | matchStatement | iterateStatement |
                            sequenceStatement | parallelStatement | scopedStatementList |
                            returnStatement] ";" .
assignmentStatement     ::= identifierOrNilList ((":" (constantDecl | typeDecl | typeExpr)) |
                            (assignOp expression)) |
                            "(" identifierOrNilList ")" assignOp expression .
returnStatement         ::= "return" returnSignature .
returnSignature         ::= "(" identifier ":" expression {"," identifier ":" expression} ")" .
assignOp                ::= "=" | "^=" | "|=" | "&=" | "%=" | "/=" | "*=" | "-=" | "+=" | ">>=" |
                            "<<=" | "<-=" | ":=" .
matchStatement          ::= ("match" | "matchseq") "{" guardedStatementList "}" .
iterateStatement        ::= "iterate" "{" guardedStatementList "}" .
sequenceStatement       ::= "sequence" orderingHead scopedStatementList .
orderingHead            ::= "(" assignmentStatement ";" expression ";" assignmentStatement ")" .
guardedStatementList    ::= {(expression | chanEventExpr) "=>" scopedStatementList} .

/*
*   Expressions
*/
expression              ::= (term {lowPrecedenceBinaryOp term}) | anonAggrCastExpr | loadExpr .
arrayCastExpr           ::= "array" (("of" idxInitList) |
                            ("[" integerConst "]" "of" starInitList)) .
anonAggrCastExpr        ::= arrayCastExpr .
loadExpr                ::= "load" identifier ["->" identifier] [tupleType]
                                (stringConst | "(" "path" typeName ")") .
term                    ::= [basicType] [unaryOp] factor ["++" | "--"]
                                {highPrecedenceBinaryOp factor} .
factor                  ::= qualifiedIdentifier | integerConst | realConst | stringConst |
                            boolConst | "(" expression ")" | namegenInvokeShorthand |
                            typeMinExpr | typeMaxExpr | "nil" .
qualifiedIdentifier     ::= identifier {fieldSelect} .
typeMinExpr             ::= "typemin" "(" numericType ")" .
typeMaxExpr             ::= "typemax" "(" numericType ")" .
namegenInvokeShorthand  ::= identifier "(" [identifier ":" expression
                                {","  identifier ":" expression}] ")" .
fieldSelect             ::= ("[" expression [":" expression] "]") .
highPrecedenceBinaryOp  ::= "*" | "/" | "%" | "&" | highPrecedenceBinaryBoolOp .
lowPrecedenceBinaryOp   ::= "+" | "-" | ">>" | "<<" | "|"  | cmpOp | lowPrecedenceBinaryBoolOp .
cmpOp                   ::= "==" | "!=" | ">" | "<" | "<=" | ">=" .
unaryOp                 ::= "~" | "-"  | "+" | "<-" | unaryBoolOp .
lowPrecedenceBinaryBoolOp       ::= "||" .
highPrecedenceBinaryBoolOp      ::= "&&" | "^" .
unaryBoolOp                     ::= "!" .
```

Programs

Programs in Noisy consist of a single program interface definition (*module definitions*) and a collection of functions. Since functions can also be used as channel specifiers and can be accessed using channel notation, the Noisy Language Specification[18] also uses the term *name generators* for functions in Noisy. This happens because both channels and functions are conceptually entries in a name space. Module definitions can specify a set of constants, type declarations and publicly-visible functions (name-generators).

Program structure

In Noisy programs consist of *modules*, their implementations and a collection of functions. A *module* is a unit of modularity that defines a set of types, constants and functions. The unit of compilation is a single module and the implementation of its functions. A function is a collection of type declarations, variable definitions and statements. Functions do not share state between them (e.g. global variables). Apart from functions of a module there can be other functions that are local and their definition is together with their implementation. On the contrary module function's definition is separated from their implementation. A program can also use the module definition of another module and dynamically load its functions and use them.

Variables and Channels

There are two kinds of program-defined identifier in Noisy: *variables* and *channels*. Variables are identifiers that are used to refer to possibly-structured data in memory. Variables are declared to have a specific Noisy type and only values of that type can be assigned to them. We can interact in two ways with variables either by reading their value or by writing (assigning) a new value to them. More details about the types will be given in a next subsection.

Channels on the other hand are identifiers that are used by the program to refer to communication paths between sequential processes (functions). Channels are a communication mechanism that trace its roots to Hoare's communicating sequential processes (CSP) [32], which is described in more detail in Chapter 2. Channels in Noisy are similar to other implementations of languages such as Alef [19], Limbo [20], Go [21] and continuation variables in Cilk. Interaction between functions and channels is done using the *send* and *receive* operations and their corresponding operators (->, <-). *Send* operation writes to a channel, while *receive* reads a value from the channel. Channels in Noisy are one-way, typed, unbuffered and synchronous. One-way means that the sender who uses a channel cannot use the same channel to receive values. The same applies to the receiver who cannot use the channel, used to receive values, in order to send back. Typed means that channels are declared to have a specific data type and thus only values of that type can be communicated using the channel interface. Type errors of channels are detected by the compiler. Unbuffered means that only one item can be sent through a channel and thus the channel does not operate as a buffer where the sender can send multiple values at a time. Synchronous means that a send (receive) completes when a matching receive (send) is performed at the other end of the channel, making the channel communication a synchronization point for the program. This communication is analogous to the producer-consumer model, in which the producer can produce the next value only when the consumer has consumed the previous. This model of communication is the opposite of buffered and asynchronous communication where a sender can send values continuously without the receiver having to receive them at the same time.

Scopes

Scopes in Noisy are defined similarly to other languages and are the part of the program where a name binding is valid, that is where the name can be used to refer to a specific entity. This means that variables defined within a scope have visibility only within the given scope. Each function defines a new scope for its body. Also there is no global scope in Noisy. Other grammar productions that introduce new scopes are:

- Module bodies.
- ADT type declaration bodies.
- Scoped statement lists (lists of statements between **{** and **}**).
- Guarded scoped statement lists as defined in the grammar.

Types, type expressions and type declarations

Noisy is statically checked and strongly typed. This means that every expression can be assigned with a unique type and programs are not accepted during compile-time if a type error can occur at runtime. Noisy features the basic arithmetic data types such as **bool**, **nat4**, **nat8**, **nat16**, **nat32**, **nat64**, **nat128**, **string**, **int4**, **int8**, **int16**, **int32**, **int64**, **int128**, **float4**, **float8**, **float16**, **float32**, **float64**, **float128**. Booleans are 1-bit values, nat-n are n-bit unsigned integers, int-n are n-bit signed integers in two's complement format and float-n are n-bit floating-point values following the IEEE-754 standard. Also there is a representation for fixed-point real values using **fixed** type. Fixed values have form $m.n$ and are signed (m+n+1)-bit binary fixed point values where m bits are used to represent the whole component, n bits are used to represent the fractional component and one bit for the sign. Also there is the **rational** type for rational numbers. Representation of values with a small number of bits (4-bit values) makes sense in memory constrained devices, because in many programs there are variables that use a small value range or flags that could not use the whole range of values of a bigger type. Variables in Noisy can have associated with them designators for channels, signals, significant figures and channel *error*, *loss*, or *latency* tolerance constraints.

Strings are sequences of Unicode characters. We can obtain a character element from a string using the notation *string_identifier*[index]. Also we can concatenate strings using + operator or query the length of a string using **length** operator. The empty string is equivalent to value **nil**. We can also obtain a substring or a *slice* of a string using the notation *string_identifier*[start_index(optional)':' end_index(optional)].

As we mentioned before types can also have *dimension designations* corresponding to the seven base SI units or one of five derived signals. Furthermore, types can have units of measure associated with them also for the base SI units and the derived signals. All those dimension or unit designation can be enriched using Newton[5] specifications that specify new dimension names, signals or units of measurement.

Noisy also features type collections like **arrays**, **lists**, **vectors**, **abstract data types (ADTs)**, **tuples** and **sets**. Arrays are constant-size sequences of elements of the same

type. Lists are also collections of elements of one type but with access only to the first element. Tuples and ADTs are collections of elements of possibly different types; whereas an ADT collection has an associated type name, tuples are unnamed collections. Sets are unordered collections of elements with primitive operations for union, intersection, relative complement and cardinality.

Arrays store sequences of items of a single type. Multi-dimensional arrays are stored in memory in row-major format and an array's length is given at the point of its definition, thus there is no array declaration and array definition as there is in some languages. In order to obtain an element from the array *array_identifier*[index] notation is used. Also the **length** operator applied to an array yields the length of the array. A *slice* of an array (sub-array) cab be obtained using the notation:
array_identifier[start_index (optional) ':' end_index (optional)].

Vectors in Noisy, unlike other languages like C++ are not collection types, but rather are Euclidean vectors. They are geometric objects in a n-dimensional space and denote a direction. They also have specific operators for vector-operations like dot and cross product.

A **tuple** is an unnamed collection of items of possibly different types. A tuple can be considered as a cartesian product of type expressions. The elements in a tuple cannot be accessed individually. This means that assignment to a a tuple must be from a tuple expression and assignment from a tuple must be to a tuple variable. Thus any assignment to a tuple sets all fields.

Aggregate Data types (ADTs) are similar to tuples, with the only difference being that the members of an ADT may be named. This permits fields of ADTs to be accessed individually. Also assignment from tuples to ADTs with the same order and type of variables is permitted. ADTs declare a new type name and therefore variables can be declared using this type name, in contrast to tuples whose type is unnamed.

The **set** collection data type is used to represent unordered collections of data items of the same type. The operations defined on set variables are set union ($A \cup B$ is denoted by A + B), set intersection ($A \cap B$ is denoted by A & B), set relative complement ($A \cap \overline{B}$ is denoted by A - B) and set cardinality ($|S|$ is denoted by **length** S). Also there is conversion (cast) from a set to a non-deterministic ordered list and a cast from list to a set. Multiplicity of elements is ignored and also an empty set has the value **nil**.

Lists in Noisy are also collections of items of a single type. However, unlike arrays accessing is only possible for the first element of the list using the **head** operator. Also another operation on lists is **tail** which yields a list containing all elements of the original list apart from the first element. Cons (::) operator is also used to append an element to the head of the list. Finally, **length** operator is used to yield the number of elements in a list.

Statements

- **Assignment statement**: Assignment statements are statements with an r-value and an l-value separated by an assignment operator. The assignment operator is =

which can be combined, with other operators and create more assignment operators. Both r-value and l-value must have the same type for an assignment to be well-typed, with the exception of value **nil**, which can be assigned to any type. When the l-value has value **nil**, then r-value is evaluated but the assignment does not yield any further results.

- **Expressions**: Expressions are syntactic entities that are evaluated to determine their value. *Expressions* are made up of *terms* and the low precedence binary operators; terms are made up of *factors* and the high precedence operators; factors can be *l-values*, constants, expressions in parenthesis or unary operators followed by a factor. Unary operators thus have the highest precedence, followed by high precedence binary operators and, then low precedence binary operators. The type of expressions must be determined during compilation, since they evaluate to a single value.

- **The match / matchseq constructs**: The match statement is a collection of guarded statement blocks. It executes *all* constituent statement blocks whose guards, which are Boolean expressions evaluate to true. If multiple guards evaluate to true, the guarded statement blocks are evaluated in the order their are written on the program. The **matchseq** statement evaluates sequentially its guards, until a guard evaluates to true. The guarded statements of this guard are then executed and the **matchseq** statement completes. These statements can be used to implement the equivalent of **if** and **switch** statements of C language.

- **The iterate construct**: The **iterate** statement is a repetitive collection of guarded statement blocks. It executes repeatedly while any of its guards, which are Boolean expressions, evaluate to true. All statement whose guards evaluate to true execute. The **iterate** statement is analogous to *guarded selection* in Djikstra's guarded commands[4].

- **The sequence and parallel construct**: The **sequence** statement is the equivalent of **for** in C language in terms of syntax and semantics. The **parallel** statement is a concurrent iteration construct.

Operators

Operator	Decription
.	ADT member access
[]	Array or character string subscript
!	Logical not
~	Bitwise not
++	Increment
−−	Decrement
load	Load function instance, instantiate a channel
type	Type cast
*,/,%	Multiplication,division,modulo
+	Unary plus,addition, set union, string concatenation
−	Unary minus, subtraction, set difference
«	Logical left shift
»	Logical right shift
**	Exponentiation
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equals
!=	Not equals
&	Bitwise AND, set intersection
^	Bitwise XOR
|	Bitwise OR

Operator	Decription
::	List append
&&	Logical AND
||	Logical OR
=	Assignment
:=	Declaration and assignment
+=	Addition/concatenation/union and assignment
-=	Subtraction/difference and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Modulo and assignment
&=	Bitwise AND and assignment
|=	Bitwise OR and assignment
^=	Bitwise XOR and assignment
«=	Logical left shift and assignment
»=	Logical right shift and assignment
<-	Unary write to channel
<-=	Read from channel and assignment
head	List head value
tail	List tail value
length	List,array,string length; set size
sort	List,array,string; sort by **valfn**
reverse	List,array,string; reverse by **valfn**
typemin	Minimum value of an arithmetic type
typemax	Maximum value of an arithmetic type
typeof	Type of argument

4.2 Noisy and Newton

The Newton language (Lim & Stanley-Marbell, [5]) is a specification language for describing physics. During the early conception of Noisy, Newton was considered an important tool for the Noisy compiler and thus they share a lot in their compiler infrastructure. More specifically, Newton is targeted to hardware and sensor designers as a specification language that could describe the physical properties of hardware (e.g. measurements of sensors). This information could be parsed by the Newton infrastructure and create specific data structures that could later be used by the Noisy compiler. Also, Newton specification files could describe information about physical signals, dimensions, or units of measure that could be incorporated in the Noisy's type system to enrich the basic type system that Noisy already had and Newton specification files could be queried during compilation of a program. Also, Newton specification files regarding sensors could be parsed in order to create channel interfaces through which a program could communicate with a sensor. Therefore, Newton is a very important part of the Noisy compiler's infrastructure.

Despite the original design for Newton and Noisy, Newton language was developed further as a standalone tool. Newton specification files could be parsed and used by a set of backends for different purposes. In particular, some of the use cases of Newton are in Dimensional Function Synthesis, which is a new method for automatic model inference between multiple signals of a physical system (Wang et al., [22]), in circuit synthesis as shown in (Tsoutsouras et al., [23]) by using the same method and in automatic synthesis of Extended Kalman Filters as shown in (Kaparounakis et al., [24]). In the same work, they also show a method for automatic derivative calculation using the compiler of Newton that improves the performance of the auto-generated filter. Thus, even though Newton is still in its infancy it has shown great potential in the field of physical system description and computation.

Noisy and Newton, share a lot in their compiler design and implementation, in order for them to interoperate as easily and effectively as possible. However, the aforementioned

functionalities between the two compiler implementations have not yet been implemented since Noisy's development was at an earlier stage compared to Newton. As a result, after our work in the Noisy compiler, future work could be done in order to investigate the core ideas between Newton and Noisy and how they could work together.

Chapter 5

Design and Implementation

Noisy has a rich type system, constructs for basic and more complex operations and a unique set of statements as described in Chapter 3. Although the Noisy specification has many and diverse features for embedded system domain, only the lexer and the parser for the language was implemented. In Stanley-Marbell's work, there is a compiler frontend implementation for the whole language specification that recognizes syntactically correct programs and prints an AST in graphviz's dot format [25]. The whole compiler infrastructure, which has also parts shared with the Newton compiler is written in C language. Our implementation starts from the previous implementation, fixes bugs, adds a semantic analyzer and finally adds a code generation step. However, since the language specification is very large we chose a smaller subset of the original language. This subset consists of all the basic statements of the computational core of Noisy for all basic singular data types as well as arrays. Also, we experimented with the channels and the CSP model of the language, and created a foundation for a complete implementation in a future work. For the code generation we chose the LLVM (Low Level Virtual Machine) compiler infrastructure and its tools.

5.1 Previous implementation

In this Section we will cover in short the previous implementation which we built upon. More details for the issues covered in this section can be found in the Implementation Chapter of the Noisy language reference[18].

5.1.1 Abstract Syntax tree

As described in Chapter 2, a top-down compiler creates a semantically equal, in-memory representation of the compiled program in the form of a tree called the Abstract Syntax Tree. The lexical analyzer reads the input source program into a sequence of tokens. These tokens can be reserved words of the language, operators, separators or identifiers. These tokens are then consumed by the parser and an AST is created. Also, the lexer stores information about each token recognized such as its position in the original source file, used for better error messages. The parser is structured as a set of functions, one per grammar production. The parser starts with the start symbol of the program and recursively calls the specific function that corresponds to the grammar production determined by the input

tokens provided by the lexer. Since Noisy's grammar was designed to be LL(1), the process of choosing the correct production is based on the *FIRST(X)* set as described in Chapter 2. *FOLLOW(A)* set is used for error recovery when a syntax error is found. These two sets are automatically generated from the original grammar specification using the Wirth tools [26].

The AST is a binary tree, meaning that each node has at most two children. If we want to represent a node with more children than two, the children are hung off in the right subtree of the parent node using "XSeq" helper nodes for chaining. Tree nodes contain references to symbol table entries and to type trees, if relevant. For many productions of the grammar they chose to shorten the height of the AST by removing unnecessary children from the nodes. The functionality of the AST is not affected and the task of functions that traverse the tree is greatly simplified, with this technique.

5.1.2 The symbol table

The symbol table is a data structure that holds semantic information about identifiers found in a program. The semantics of an identifier in an expression cannot be determined alone by the syntactic rules. Therefore, information like its type, its properties (e.g. whether it is a function name, a module name, a variable name) must be saved in the symbol table. Symbol table is used to check implicit variable declarations, find undeclared variables and type errors. Their implementation for the symbol table is a generalized tree with each subtree corresponding to a *scope*. In each scope node there is a reference to a doubly linked list of identifiers, corresponding to the identifiers of that scope. List items also contain semantic information like the type tree of an identifier.

5.1.3 Types

Types are represented in the symbol table as trees built out of the primitive types and collection type constructors. The type trees correspond to the parsed trees of type expressions. Type expressions contain information both for the basic types (e.g. int, float) and (possibly) for dimensionality and units of measure. For modules, the ordering of function definitions does not affect the module's type signature. For the rest of type expressions, two type expressions are considered equivalent only if their type trees are structurally identical. Also, the compiler performs a post-order walk of a type tree so as to create a *type signature*, unique to that type. Type signatures representations can be found in the generated graphs textual renderings.

5.2 Semantic analysis

Semantic analysis is an Abstract Syntax Tree traversal algorithm that identifies errors that cannot be identified by syntactic analysis, such as type errors, undeclared variables etc. Semantic analysis is a necessary step before the code generation phase in order to avoid runtime type errors. Semantic analysis could be performed together with the code generation phase in the form of type checking before the code generation for each tree

node but we chose to implement it as an independent AST pass for clarity reasons. Our semantic analysis implementation has similar structure with the parser, using a post-order walk of the tree and a function for each production of the grammar.

5.2.1 NoisyType data structure

Section 4.1 shows that *type trees*[18] are used in order to represent the type of an identifier and this information is stored in the symbol table. Type trees can represent a *basic type*, an *aggregate data type*, or a *typename*. The grammar defines that basic types are all the singular basic types (integers, naturals, floats, Boolean, string), aggregate data types are the collection data types mentioned in Chapter 3, and typenames are names given to types (similar to typedef in C). In our implementation we support type checking for all basic types and arrays.

Since types were expressed in the tree form and type comparisons would require a tree traversal, we decided to simplify the process of type check. We created a function that traverses a given type tree and returns a struct called NoisyType which contains all the necessary information for type checks. Then the NoisyType object is stored in symbol table and is used for type comparisons. The NoisyType data structure is shown in Listing 5.1.

Listing 5.1: NoisyType data structure.

```
typedef struct NoisyType     NoisyType;

enum
{
    kNoisyStaticArrayMaxNumberOfDimensions = 128
};

struct NoisyType
{
    NoisyBasicType basicType;
    int dimensions;
    NoisyBasicType arrayType;
    Symbol * functionDefinition;
    int sizeOfDimension[kNoisyStaticArrayMaxNumberOfDimensions];
};
```

NoisyBasicType data structure

NoisyBasicType data structure is the type of the *basicType* field of NoisyType and is shown in Listing 5.2. NoisyBasicType is an enumeration that represents all basic data types of Noisy, as well as some special values like *noisyNilType*. All NoisyBasicType objects are initialized with the value *noisyInitType*. Value *noisyTypeError* is used to represent type errors and *noisyNamegenType* refers to values that load a function or channel implementation using the **load** keyword. Values *noisyIntegerConstType*, *noisyRealConstType*, and *noisyArithType* are used to implement the type cast rules and type conversion of constant values.

Listing 5.2: NoisyBasicType data structure.

```
typedef enum
{
    noisyBasicTypeInit,
    noisyBasicTypeBool,
    noisyBasicTypeInt4,
    noisyBasicTypeInt8,
    noisyBasicTypeInt16,
    noisyBasicTypeInt32,
    noisyBasicTypeInt64,
    noisyBasicTypeInt128,
    noisyBasicTypeNat4,
    noisyBasicTypeNat8,
    noisyBasicTypeNat16,
    noisyBasicTypeNat32,
    noisyBasicTypeNat64,
    noisyBasicTypeNat128,
    noisyBasicTypeIntegerConstType,
    noisyBasicTypeFloat16,
    noisyBasicTypeFloat32,
    noisyBasicTypeFloat64,
    noisyBasicTypeFloat128,
    noisyBasicTypeRealConstType,
    noisyBasicTypeArithType,
    noisyBasicTypeString,
    noisyBasicTypeArrayType,
    noisyBasicTypeNilType,
    noisyBasicTypeNamegenType,
    noisyBasicTypeErrorType
} NoisyBasicType;

```

Array type representation

The *NoisyArrayType* represents the Noisy array data type but it is not enough since it does not contain information about the type of the elements of the array, its dimensions and their size. Thus, we use *dimensions* field for the number of dimensions of the array, *arrayType* field for the type of the items stored in the array, and *sizeOfDimension* field for the size of each dimension of the array. The *sizeOfDimension* has 128 elements, because it was easier to use a static array, since it is highly unlikely for a 128-dimension array to be used in Noisy. All those fields are necessary for Noisy's array type representation because arrays are statically allocated and their size is always given at compile-time .

Noisy constant types

Constants in Noisy are defined in the grammar either as Boolean, integer, float or string. Thus, the size of their type is not given explicitly (for integer, floats, and strings). To represent that, we have used the values *noisyIntegerConstType*, *noisyRealConstType*, and *noisyArithType* which are super sets of integer and float types. When a constant is encountered in an expression in Noisy, its type is assumed to be the type needed for the correct evaluation of the expression. Therefore, we have implemented a small type inference mechanism for constants, that traverses an expression tree and infers the type of the constants found in it. In order to implement the type inference mechanism we assign

NoisyType values to expression IrNodes. If a constant is encountered the NoisyType of its parent node is checked. If it has a NoisyType that is more "specific" than the type of the constant then the constant is inferred to have this type. This recursion ends when it reaches the expression node. In the current stage of the implementation it is not permitted for an integer constant to be inferred as a float, unless there is an explicit type cast.

Name generator types

When a function implementation is loaded using the **load** operator, it is assigned to a variable identifier. This identifier has type *noisyNamegenType*, however it does not exactly work as a variable. This identifier stores a reference to the implementation for a function and this function can be called using this identifier instead of the original function name. In the noisyType data structure the *functionDefinition* field stores a pointer to the specific symbol table entry that represents the exact function implementation and it is used for the function's type checking. As we will see next, name generators can store the implementation of a specialized function (from templated types), or a coroutine object.

Type equality with NoisyType

We define that two types are equal when their *basicType* values are equal. In the case of *noisyArrayType*, *arrayType*, *dimensions*, and *sizeOfDimensions* values must be also equal. For *noisyNamegenType* we use *functionDefinition* to determine type correctness based on the context and function semantics rules.

5.2.2 Semantic Rules

Expressions

Expressions need to be well typed in order to be accepted by the semantic analysis. Each well typed node of the expression tree (*terms* and *factors*) evaluates to a single type and by using type rules the expression also evaluates to a single type. Type errors interrupt the compilation of the given program and return an error message indicating the error and its position in the source file. The type rules we have implemented are presented below:

- **Variable identifiers**: Variables have their type declared and their type is found by searching the symbol table. If an identifier is not in the symbol table then compilation terminates with an "unknown identifier" error. Multiple declarations of the same identifier in the same scope are not permitted. Variables that have been loaded using the **load** operator follow the rules for function identifiers. If a variable is also a channel then it can be used only with the channel operators for writing and reading from/to a channel.

- **Function identifiers**: Function type is given by two type *signatures*; the input signature and the output signature. The input signature is a comma separated list of named parameters, thus a parameter is actually a parameter name and its type. The output signature can syntactically have the same form as an input signature. In

our implementation, however, we chose to permit only one return argument and as a result, the output signature is a single parameter (its name and its type). Also, either or both signatures can have type nil (*noisyNilType*). When nil value is in the input signature means that the functions takes no parameters as arguments and when the output has nil signature means that the function returns void (does not return a value). Function identifiers found in expressions are function calls. Since function parameters in Noisy have names, their order is irrelevant. Formal and actual parameters are type checked one by one based on the name of the parameter and must be equal. A function call with more or fewer arguments than those declared is considered a type error. Also, if a formal parameter's name does not exist, it is also a type error. If there is not a type error in the input signature, the function identifier evaluates to the type of the output signature.

- **Binary operators**: Both operands of a binary operator must be of the same type, with the only exception being constants which can have a type super set. Constants, however, are inferred to have the same type with the other operands during code generation. Arithmetic and Boolean binary operators (+,-,/,*,**,||,&&) evaluate to the same arithmetic or Boolean type as their operands, while comparison operators (>,<,>=,<=,==,!=) evaluate to Boolean type. Modulo operator (%) accepts only integer or natural operands. Bitwise binary operators (&,∧,|) accept arithmetic operands and return the type of their operands. Shift operators («,») accept arithmetic type as the left operand and natural type as their right operand.

- **Unary operators**: Unary minus (-), applies to integers and floats and returns the same type as its operand. Negation (!) accepts only a Boolean operand. Bitwise not (˜) accepts only an arithmetic type operand. Unary channel read operator's (<-) operand must have name generator type (*noisyNamegenType*) and also the function implementation must be a channel. The returned type is evaluated to be the output signature parameter type of the function (the type of the output channel). Increment and decrement (++, –) operators accept only integer and natural operands and return the type of their operand.

- **Type casting**: Type casting is a special unary operator that converts an expression from one type to another, based on a set of permitted conversions. Type casting is allowed only for arithmetic type arguments (integers,naturals, floats). More specifically, all types can be converted to the same type but with a greater number of bits (extension), or to the same type with less number of bits (truncation). Also conversions from naturals to integers, from integers to floats and vice versa are permitted.

Statements

Statements do not evaluate to a type like expressions, but there are statements that must obey specific semantic rules to be correct.

- **Assignment statement**: Assignment statements assign the value given as an expression on their right hand side (RHS) operand and assign it to the left hand side

(LHS) operand. Both those operands must have equal types. The simple assignment (=) as well as assignment and declaration (:=) can be performed for values of all types (even for arrays). For assignment and declaration if the RHS operand evaluates to a type super set (*noisyArithType*, *noisyRealConstType*, *noisyIntegerConstType*), assignment will fail resulting in a type error because the compiler will not know what type should be given to this value. The rest of the assignment operators (+=, -=, *=, /=, %=, &= etc) must obey the rules for both assignment and their second operator (e.g. %= operator can accept integer or naturals operands). A special case of assignment is the write to channel assignment operator (<-=) witch accepts a name generator type (*noisyNamegenType*) and channel as the RHS operand and the type of LHS operand must be equal to the type of RHS operand's input channel.

- **Guarded statements**: **Match**, **matchseq** and **iterate** statements all have guarded statements. For guarded statements, the guard expression must evaluate to Boolean type.
- **Sequence statement**: The second of the three expressions in the **sequence** syntax must evaluate to Boolean type. Also the parenthesis of the sequence statement does not create a new scope.
- **Return statement**: At least one return statement must exist in functions with a return type other than nil. In this case, the expression of the return statement must have the same type with the output signature of the function.

5.2.3 Type templates

Noisy offers a mechanism for type templates, where the user can write a function for an abstract data type and then the function's implementation can be specialized for a specific data type using the **load** operator. This feature, allows programmers to write a single function definition that can be used with a variety of different data types. The **module** syntax allows for a *typeParameterList* production which is a list of identifiers followed by ":" and the keyword **type**. We will refer to this data type identifier as an abstract data type. In the following example we see that *matrixDataType* is such an identifier. We see that the matrixMulConst is also a function that uses this identifier in its type signature and from now on we will refer to functions with abstract data types as abstract functions. Later, in the init function the matrixMulConst function is specialized for int32 data and its implementation can be accessed using the *int32Mul* variable identifier. This variable identifier can be used just like a function name would be used to call a function in Noisy. Also, the *matrixMulConst* name cannot be used in a function call because it is an abstract function.

Listing 5.3: Templated module.

```
1 # Templated module. The matrixDataType identifier is an abstract data type name.
2 matrix : module(matrixDataType : type)
3 {
4     # This function uses the abstract data type in its signature
```

```
    matrixMulConst : function (matrixA : array [32][32] of matrixDataType,
                     constVal : matrixDataType)
                     -> (result : array [32][32] of matrixDataType);
    init : function (nil) -> (x : int32);
}

# The implementation of the function it is not type checked unless an explicit load
# instantiates abstract data types.
matrixMulConst : function (matrixA : array [32][32] of matrixDataType,
                 constVal : matrixDataType)
                 -> (result : array [32][32] of matrixDataType) =
{
    i,j : int32;
    sequence(i = 0; i < 32; i += 1)
    {
        sequence(j = 0; j < 32; j += 1)
        {
            matrixA[i][j] *= constVal;
        };
    };

    return (result : matrixA);
}

init : function (nil) -> (x : int32) =
{
    ...
    # When a function with abstract data types is instantiated with
    # a specific data type, then it is type checked.
    int32Mul := load matrix->matrixMulConst (int32) (path matrix);
    ...
    # The arguments of the specialized function are also type checked
    # against the specialized version of the function.
    matrixA = int32Mul(matrixA : matrixA,constVal : 2);
    ...
}
```

Implementation

Functions in the Noisy compiler are represented as subtrees of the program's AST. Apart from this representation, function's name, parameters and identifiers are saved in the symbol table. Abstract functions differentiate from other functions only because they have an abstract data type in their type signature. Therefore, our semantic analyzer does not type check abstract functions' definition, unless a type instantiation has been performed by the **load** operator. This means that when a function is loaded then the abstract data type evaluates to a specific type and type checking can begin. However, we want to store both the abstract function and the specialized one because the abstract function can be instantiated again with a different type by another load operation. In order to solve this issue, we decided to clone the original function tree of the abstract function and instantiate its data types in the cloned tree. The new tree is assigned to the variable identifier that is in the LHS of the load operation and the user can access the instantiated function by using this variable identifier just as a function name would be used. In order to clone the original tree, a deep copy is necessary, meaning that we create copies of each tree node of the original abstract function instead of keeping pointers to the original nodes. This is

necessary because we do not want to alter the original abstract function tree. Also, it is mandatory to change the names of the instantiated identifiers in the symbol table, since symbols are searched based on their name and using symbols with the same name would result in altering the original tree's symbols. We append an, increasing with every load, number to each identifier's name of the instantiated function in order to achieve this.

Our implementation can not infer the accepted type values for an abstract data type based only on a function implementation (e.g by checking operators). On the contrary, types must be first instantiated for the function definition to be type checked (for the specific type). Implementing a better solution for type templates would be an interesting addition to the type template mechanism.

5.2.4 Name generators and channels

Functions in Noisy can use both the semantic conventions for functions in C (parameters as input and return statement for output) and the channel interface where a function can have many input channels as well as many output channels. Therefore, the Noisy language reference[18] also refer to functions, as name generators, to showcase that a function name in Noisy can use those two semantic alternatives. In order to differentiate a C convention function call from the channel interface from now on we will use the terms functions and coroutines, respectively. Functions are sequential parts of code that accept arguments and their execution finishes with a return statement which also returns a value to the caller of the function. Coroutines are parts of code that use the channel interface as a communication mechanism (input and output) and also use cooperative scheduling. Cooperative scheduling is the ability for coroutines to stop their execution at specific points of their execution so other coroutines can execute instead. The coroutine that was scheduled out saves its current state and can continue its execution at a later stage from the point it had stopped. For our implementation the coroutines' scheduling points are reads/writes from/to channels.

In this Section we will talk only about the semantic information that is needed for the semantic analysis. The implementation details for coroutines will be explained in Section 5.3. In order to use coroutines and channels in Noisy, the **load** operator is necessary. This operator loads a coroutine implementation to a variable identifier and then the user can use the variable to access the coroutine's channels. Channels are accessed only by using the channel operators (<-, <-=) and their semantics are described in Subsection 5.2.2. In that Section we mentioned that an identifier used with the channel operator symbols has to be *noisyNamegenType* and also to be a channel. An identifier is recognized as a channel only when it loaded a coroutine implementation, or it is the input or the output signature's parameter name. A name generator is identified as a coroutine only if it uses channel notation to read from the identifier of the input signature (input channel) or write to the identifier of the output signature (the output channel). Also, the current state of the grammar does not allow the programmer to use multiple input or output channels, as a result of how the load mechanic and its syntax works. Therefore, we permit coroutines with at most one input and at most one output channel.

The semantic rules for channel operator and coroutine's signature identifiers are : write to their output channel and read from their input channel. Any other channel operation is not valid (e.g. read from coroutine's output channel). The opposite is true for the caller of a coroutine, which can write to the input channel of the coroutine and can read from the output channel. The caller of a coroutine must first load the coroutine implementation before being able to use its channels and this operation instantiates both the coroutine and its input and output channel (if they exist). Also, every load operation that instantiates a coroutine creates a a different coroutine instance with different channels. Channels are typed and every channel operation (read/write) is type checked during compile-time. A coroutine can be terminated and its channels deallocated by assigning a **nil** value to its input channel or when the caller function terminates. In Listing 5.4 we present an example of Noisy code that uses the coroutine and channel semantics, which implements a fibonacci sequence number generator.

Listing 5.4: Coroutines in Noisy.

```
fib : module (valueType : type)
{
    fibonacci : function (nil) -> (output: valueType);
    readInt32 : function (nil) -> (out : int32);
    printInt32 : function (x : int32) -> (out : int32);
    init     : function (nil) -> (x : int32);
}

# Coroutine function without input channel
fibonacci : function (nil) -> (output: valueType) =
{
    a : valueType;
    b : valueType;
    n : valueType;

    a = 0;
    b = 1;
    n = b;

    iterate
    {
        true =>
        {
            # Write to the output channel of the coroutine
            output <-= n;
            n = a + b;
            a = b;
            b = n;
        }
    };
}


init : function (nil) -> (x : int32) =
{
    i : int32;
    n : int32;
    fibRet : int32;

    n = readInt32();
```

```
41
42      # Variable f has noisyNamegenType and is also a channel
43      # because the loaded function is a coroutine
44      # We also see that fibonacci is instantiated for int32 values
45      f := load fib->fibonacci (int32) (path fib);
46
47      i = 0;
48
49      iterate
50      {
51          i < n =>
52          {
53              # Read from the output channel of namegen f
54              # and store the value to fibRet variable
55              fibRet = <-f;
56              nil = printInt32(x : fibRet);
57              i+=1;
58          }
59      };
60
61      # When the function init terminates, all
62      # coroutines associated with it terminate
63      # and their channels are deallocated.
64      return (x : 0);
65  }
```

5.3 Code Generation

Code generation is the part of the compiler that is responsible to convert the in-memory intermediate representation of a program to a semantically equivalent executable form. For our implementation we chose to use the LLVM's compiler infrastructure to generate executable code for Noisy. Our code generator has also a similar structure with the parser and the semantic analyzer, being a post-order walk of the AST with functions for each production of the grammar. However, functions of the code generator return LLVM intermediate representation (IR) values, thus our code generator actually acts as a transformer from Noisy's AST to LLVM IR values. LLVM IR values are parsed by the LLVM's compiler tool set and generate object files (or assembly files) for a plethora of architectures and operating systems. The produced files then can be linked with the default C library and create the final executable binaries all thanks to LLVM's compiler infrastructure.

5.3.1 LLVM Toolchain

Low Level Virtual Machine (LLVM) is a collection of modular and reusable compiler and toolchain technologies. LLVM provides a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages [27]. LLVM is an umbrella project for its many subprojects such as LLVM Core libraries, Clang C/C++/Objective-C compiler, lldb debugger, libc++ library and many others. In our implementation we use the LLVM Core library which provides the code generation support needed in our project. The LLVM Core library is built around a well specified and thoroughly documented code representation called LLVM intermediate representation

(LLVM IR). LLVM IR is a source- and target-independent, strongly typed, intermediate representation language, using the Single Static Assignment (SSA) form. LLVM IR looks very similar with an assembly language, can be compiled to binaries for different target architectures by using LLVM Core library which also offers a great variety of code generation optimizations.

The LLVM infrastructure was chosen for our project for a variety of reasons. First of all, the use of LLVM allowed the fast development of our project since it is a well documented and widely used compiler infrastructure. Also, since Noisy aims to be used for embedded systems we did not want to target a very specific architecture for our first implementation but we wanted to be able to generate code that could run as easily on our desktops as well as to small computing platforms with very different architectures. The debugging process was a lot easier on our local machine than it would be on an ARM processor without an Operating System. Moreover, LLVM provides many code optimization routines (both target dependent and target independent) that we would have to implement ourselves had we not used it. In our implementation we used the C interface of LLVM's core library [6].

5.3.2 Structure of a program

In LLVM the *Module* is the basic unit of compilation, which is fairly similar to the syntactic structure of Noisy. Modules in LLVM consist of global constant declarations (e.g. constant numbers, string literals, array initializers), function declarations and function definitions. A function declaration is actually the name of the function and its type signature. Functions definitions are declarations with at least one basic block, which is the entry block of the function. Functions are comprised of basic blocks and basic blocks are comprised of LLVM instructions. Each module can have only one main function, which executes first.

Apart from the *Module*, there are three other important objects,the *Builder*, the *Context*, and the *PassManager*. The *Builder* is responsible for creating the basic blocks, arrange them and write instructions to them. The *Context* holds information about the compilation target (architecture, operating system, ABI) and is used for the transformation of LLVM IR to actual assembly. Finally, PassManager can run optimization *passes* to the LLVM IR to achieve better code performance. Running multiple optimization passes can lead to greater compilation times.

5.3.3 Data types

Our implementation supports all the data types we presented in the previous Section apart from float4 and float8 variables which are not supported by LLVM. LLVM uses the *LLVMTypeRef* data type to represent the various data types it supports. Our code generator converts *noisyType* values to *LLVMTypeRef* values using a specific function. This process is facilitated by the structure of noisyType that can easily be converted to an LLVM type because every arithmetic *noisyBasicType* value corresponds to a specific LLVMTypeRef value. Both natural and integer numbers of Noisy are converted to LLVM integers, since LLVM does not make a distinction between singed and unsigned integers,

although sign must be taken into account during operations. Booleans are represented using 1-bit integers and strings are not supported in the current implementation since their manipulation needed library support and more design. Arrays are statically allocated and have their size declared at compile-time. At the current stage array elements are saved in consecutive positions in memory using row-major format. Array size is not part of its memory representation (used for dynamic bound checking), meaning that the user can have an out-of-bounds indexing runtime error. The rest of noisyBasicType values do not convert to actual arithmetic data and are handled differently based on the semantics of each value. For example *noisyNamegenType* translates to a function call and actually has the type signature of that function.

5.3.4 Functions

Functions in LLVM have a name, an ordered list of unnamed parameters and a return type. All basic types are passed by value to functions. Since parameters in Noisy are named and thus their order is not important, the conversion to LLVM functions uses the parameter order of the function definition (and declaration). On function calls, values of the named parameters are matched to the correct position of the corresponding LLVM function and then the function is called.

Noisy's main function is called "init" as the language reference demands. Since LLVM uses "main" name for its the main function of a module, in our implementation we change the name of "init" function to "main". Only one "init" function can exist in a Noisy module.

Array function parameters is an interesting part of our implementation. Noisy does not provide pointers to the user, however the LLVM implementation for arrays forces arrays to be passed by reference (passing the pointer of array's start to the function). Since, all values in Noisy are passed by value and arrays were the only data types that were passed by reference, we thought that this created a discrepancy in the implementation and it could seem counter intuitive for the user. As a result, we decided that array parameters should be passed "by value". This means, that when a function is called with an array as a parameter, a new array is allocated on the caller that copies the elements of the original array. This array copy is passed by reference to the called function and as a result the original array is not mutated from the function call. This process is performed for every array parameter in a function call.

Since we chose this approach for array parameters, we wanted a mechanism for functions to return arrays. Semantics of the language allowed us to assign or return arrays. The implementation of assignment used only a memcpy operation but the return was not so straight-forward as arrays are stack allocated. Arrays allocated in a function could not be returned to the caller of that function, because they would be deallocated at function's return. In order to solve this issue, when a function returns an array variable, we allocate memory on the functions's caller. This array , called the return array, is passed by reference as an extra argument to the function and the function returns void instead. As a result, upon return, the function uses memcpy to copy the array it would return to the return

array, which is stored on the caller's stack.

Passing arrays to functions by value, uses twice the memory due to the copy mechanism. Also, the language lacks the ability to pass arrays by reference which is, on many occasions, is a useful tool. This problem could be solved by array slices, which we did not implement. Array slices do not have a constant nor known size at compile-time and thus could be passed by reference. This way, Noisy could feature both mutable and immutable array parameter pass, similar to the array implementation of Rust language [28].

5.3.5 Expressions

Expressions evaluate to a single value of a specific type. LLVM has its own value data type called *LLVMValueRef*. Every LLVMValueRef is typed in the LLVM and corresponds to a hypothetical register, since LLVM IR makes the assumption that it has an infinite number of hypothetical registers. These values are used as the operands of basic operations, each operation (in SSA form) returns a value, and values are stored or loaded to/from memory. LLVM offers support for all basic and more complex operations such as addition, subtraction, multiplication, division, shifting, exponentiation, store, load. An interesting case of LLVM operation is the getElementPointer instruction (GEP) that is used to index array and struct elements.

Code generation for expressions is the process of transforming the AST expressions to a series of computations that use LLVM values and operations. Since, LLVM uses the SSA form, each AST node is parsed following the precedence and priority of Noisy's operators, and for each operation a matching LLVM IR instruction is created. All operations require semantic information given by the *noisyType* data structure, as all operations of LLVM are typed. Some operations are easily translated to LLVM IR such as load and store, while for other operations we have to inquire extra information from the noisyType in order to choose the appropriate operation. For example, we have to choose between signed or unsigned integer operations, choose between extension on truncation by comparing type sizes for type casting and use multiple GEPs to index arrays correctly. In general, creating LLVM IR from expressions is pretty straightforward, mainly because we implement the basic types and operations.

5.3.6 Statements

Statements except for the assignment statement, change the control flow of the program. LLVM uses the concept of basic blocks (*LLVMBasicBlockRef* data type) to create a control flow graph of the program. Basic blocks are uninterrupted sequential parts of LLVM instructions. Each basic block is defined by a label and must have a terminator instruction at its end. Terminator instructions are conditional or unconditional jump instructions or the return instruction. Jump instructions are used to transfer control flow from one basic block to another (within the same function), using the basic block's label, while return instruction transfers control flow to the caller of a function.

Code generation for the assignment statement produces the store instruction, that stores a value to the memory. Assignment cannot translate to a store for arrays and name

generator types. Assignment for arrays uses the memcpy intrinsic of LLVM that copies consecutive elements in memory to another memory location (copies elements of one array to another). The implementation of assignment for name generators and channels will be analysed in the coroutine Section. Also assignment can create other instructions for the most complex assignment operators (+= , -=, etc). Return statement is also fairly simple and corresponds to the ret instruction of the LLVM.

The rest of the statements create and connect basic blocks, forming the control flow graph. Figures 5.1, 5.2, 5.3, and 5.4 present the control flow graphs for each statement we implemented, which show the basic blocks and the LLVM instructions needed for the implementation.

Figure 5.1: *CFG of a match statement.*

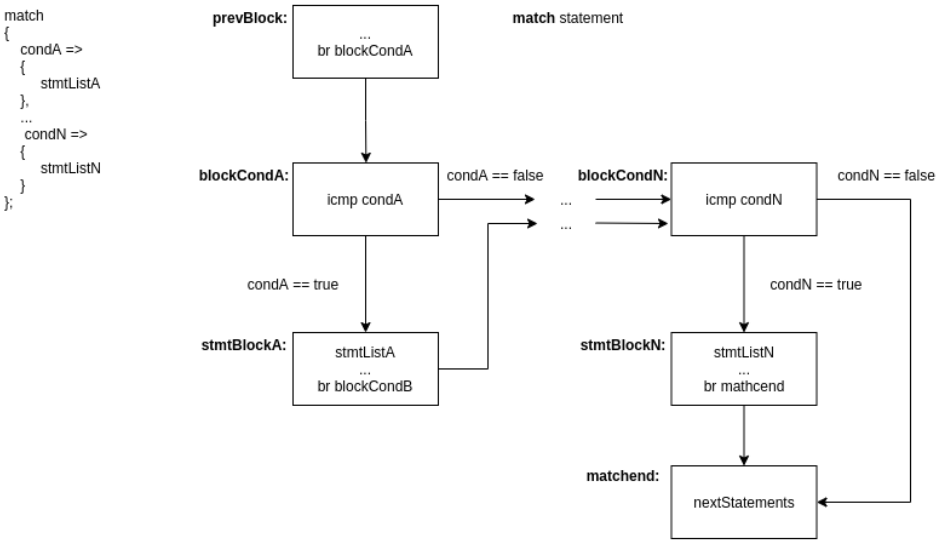


Figure 5.2: *CFG of a matchseq statement.*

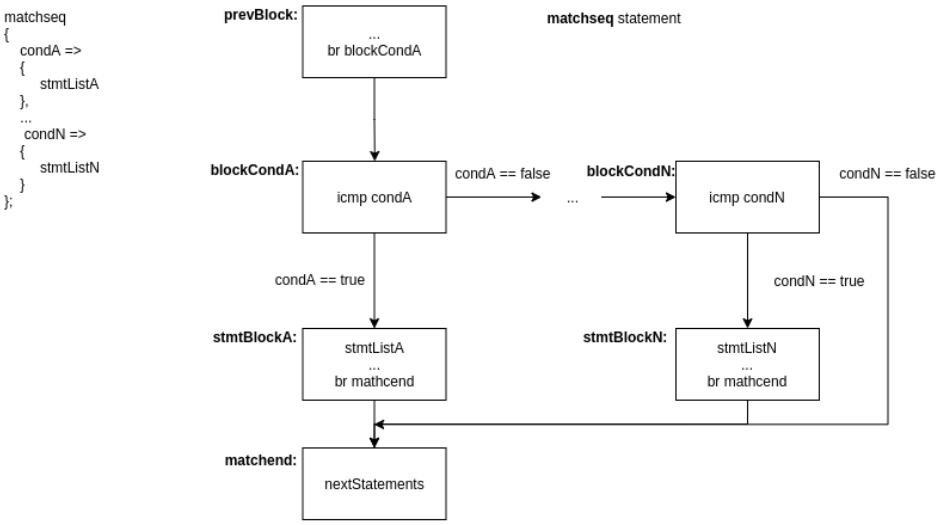
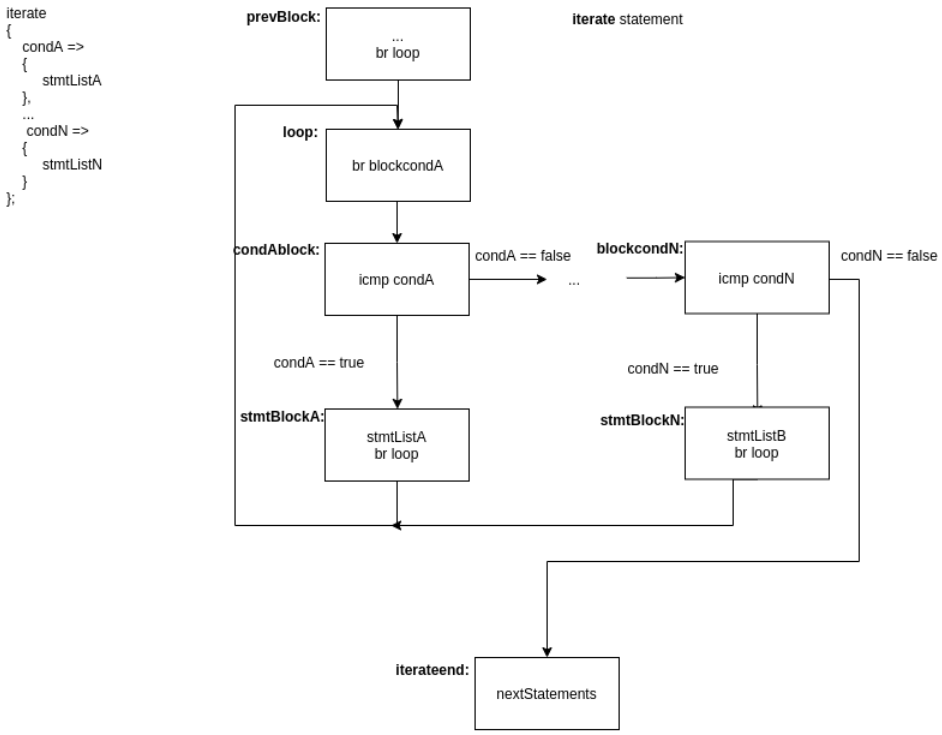


Figure 5.3: *CFG of an iterate statement.*



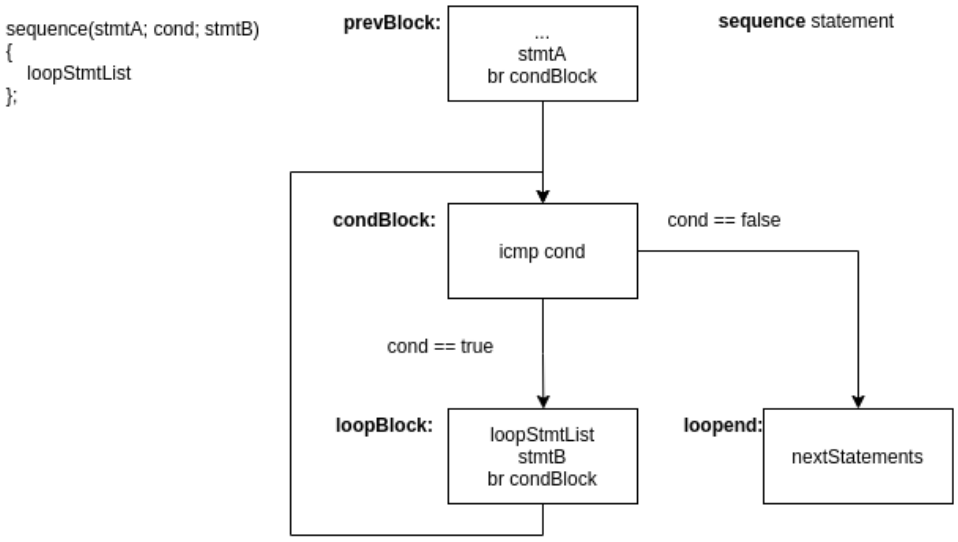
5.4 Coroutines and Channels

In Section 3.2 we explain the theoretical background for Communicating Sequential Processes. In order to implement the CSP model, there are two topics that need to be covered; the processes and their scheduling and the communication mechanism. Noisy language reference[18] defines that name generators can be used as coroutines and thus act as processes of the CSP model. Also, every coroutine can have channels for input or output which act as the communication mechanism. In this Section we will elaborate on the CSP model design and on the parts we implemented.

5.4.1 Runtime system

Since the CSP model features high-level constructs such as channels and process scheduling, a runtime system implementation is needed for their execution. The runtime system is not currently implemented but it should be implemented as a static library that is linked with every Noisy program and handles coroutines' scheduling and channels. The final goal for Noisy would be to support a M:N threading model. This threading model gives the ability for the user to create M lightweight threads (in our case coroutines) and then these should be scheduled to N OS threads for execution. This way, the runtime system acts as an interface between lightweight user threads and heavy kernel threads. The runtime system should be also responsible for interacting with the OS. Our current implementation hopes to set the foundations for implementing a complete runtime system, by using the Coroutines implementation of LLVM, which could be used as the lightweight threads needed for M:N threading model.

Figure 5.4: *CFG of a sequence statement.*



5.4.2 Coroutines

Processes of the CSP model must be lightweight and should implement cooperative scheduling. User lightweight processes should be created by the user in large numbers as opposed to heavy kernel threads, which are limited based on the hardware that a program is currently running. These processes are a tool for programmers to reason better about their program and its functionality and they are not an actual resource of the program. Runtime system should be responsible for the scheduling of those processes. The most important parameter is the cost of context switching for lightweight processes. It becomes apparent that neither processes nor OS threads can be used to implement coroutines because they are limited and also the cost of their context switching is very high, because the whole processor state must be saved. In order for an implementation to be effective, the cost of context switching should be similar to a function call. In (Pettersen, [29]) they show a coroutine implementation using C functions and inline assembly for context switching. However, an inline assembly solution would require porting this solution to different hardware, which contradicts our reasoning about the portability of our implementation and also we wanted to investigate the LLVM's tools, since we were already using this toolchain.

5.4.3 LLVM Coroutines

LLVM's coroutines [7] is a new feature of the LLVM Core library that was used for C++20 coroutines. Coroutines in LLVM are functions with suspend points. Their semantics dictate that when a suspend point is reached, the execution of a coroutine is suspended and control is returned back to its caller. Also, a suspended coroutine can continue execution from its last suspend point or can be destroyed. Coroutines in LLVM have a stack frame when they are executing and also an additional region of storage which is called the **coroutine frame**. Coroutine frame stores the state of a coroutine when it is suspended. LLVM offers a set of different coroutine intrinsic functions that can be generated by the

compiler. These coroutine intrinsics represent the main coroutine functionalities, such as suspension, resumption or termination. These intrinsics are then used by LLVM's optimization passes, specific to coroutines, and coroutines are lowered and rewritten in a form that can be compiled to assembly. Those passes are called lowerings and each lowering features a different strategy for lowering the initial coroutine. In our implementation we use *switch-resume* lowering.

Switch-resume lowering takes the initial coroutine LLVM representation and it splits it into three functions, representing the three ways that control can enter a coroutine:

- **Ramp function**: The ramp function is initially invoked upon a coroutine call, takes arbitrary arguments, initializes the coroutine frame and returns a pointer to the coroutine frame. This pointer is used by the next functions in order to access it.
- **Resume function**: The coroutine resume function is the function that is invoked every time this coroutine is resumed. Resume function takes the coroutine frame pointer as argument and returns void.
- **Destroy function**: The coroutine destroy function is invoked when the coroutine is destroyed and thus is responsible for deallocating the coroutine's frame. It also accepts as argument a coroutine object and returns void.

This way, we can use LLVM's implementation to create Noisy's coroutines in a way that does not use macro commands or inline assembly for context switching between coroutines. Also, this way context switching adds small execution overhead (similar to a function call) since the resume intrinsic is actually lowered to a function call of the resume function. As a result, we decided to use LLVM's coroutines to implement the coroutine semantics and operations. This means that we can generate code from our code generator that uses LLVM's coroutine intrinsics, which is a relatively simple task, and then the LLVM optimization pass will guarantee that the intrinsics will be lowered to function calls and thus we will be able to directly compile this implementation into different assemblies using LLVM's static compiler.

5.4.4 Channels

Channels are the communication mechanism that a CSP model uses. We believe that channel creation, deallocation and run time error handling should be done by the runtime system. Even though we use LLVM coroutines for the processes, channels should be different objects handled exclusively by the runtime system. When we generate code for a coroutine, a specific function of the runtime should be called for allocation of the channels. The same should apply for deallocation of the channels. When a channel read/write is encountered, the code generator should generate a function call to the runtime system that should implement the semantics of read/write. Since, we did not implement the runtime system we created a proof of concept implementation of the channels that uses LLVM's coroutines intrinsics and as a result we have a working version of coroutines and channels. Our implementation of channels produces correct results for correct programs, although

it does not execute correctly when there are runtime errors (errors are not handled) at channel operations.

LLVM offers an intrinsic promise feature called *llvm.coro.promise*. Coroutines return values to their caller using the promise intrinsic. In order to emulate the input channel of a function, we create a variable that is passed by reference to the coroutines arguments. Therefore, the caller writes to that value upon channel write and the coroutine loads its value upon channel read. Since, both input and output channels of a coroutine are implemented as variables we do not have a mechanism to implement read and write failures (writing to full channel, reading from empty channel) and we expect that this operation should be handled by the runtime system. Also, synchronization on channel operations is implemented using coroutine suspend and resume. When the runtime implementation is completed suspend will return control to the caller of the coroutine, but the caller of the coroutine should be an initial coroutine, part of the runtime system that is in charge of coroutine scheduling. For now, control is returned to the function that called it and scheduling is not implemented. The details of each channel operator implementation will be explained next.

5.4.5 Channel operations

Load operator

Load operator loads the coroutine object and returns a pointer to it, which is then assigned to a variable, so that the user can handle the coroutines channels. In our implementation load operator generates the *llvm.coro.begin* intrinsic which allocates the coroutine stack frame and returns a pointer to *llvm.coro.id* object that is responsible for coroutine manipulation. Also, load operator generates code for the whole structure of a coroutine; it creates basic blocks and code for the *llvm.coro.destroy* intrinsic, which deallocates the coroutine and also generates a *llvm.coro.suspend* intrinsic to return control to the caller of the function. Semantically, **load** operator allocates the memory for the coroutine (frame) and returns a pointer to the coroutine object.

Channel write

Channel write operator (<-=) has two distinct implementations for its different use cases. The first use case is when the caller function writes to the input channel of a coroutine. In that case, we generate a write to channel operation and then we invoke *llvm.coro.resume* so the coroutine can continue its execution since a new value is written to its input channel. The other case is a coroutine writing to its output channel. In that case, we generate a channel write to the coroutine's promise and then a *llvm.coro.suspend* intrinsic so coroutine can transfer control to its caller and the caller can consume the value and continue its execution.

Channel read

Channel read operator also has two distinct implementations. When a coroutine reads from its input channel, we generate a read from input channel instruction and then a *llvm.coro.suspend* which transfers control flow back to the coroutines caller. When a function reads from a coroutine's output channel, we generate a read from promise instruction and then a *llvm.coro.resume* transferring control to the coroutine.

5.4.6 Coroutine Destruction

Coroutine deallocation can be invoked manually by assigning a **nil** value to the input channel of a function. Otherwise, when a function that had coroutines in its scope terminates, these coroutines are deallocated automatically. Deallocation, is performed by invoking *llvm.coro.resume* with the appropriate flag for destruction which in turn branches to the basic block of *llvm.coro.destroy* intrinsic, which destroys the coroutine frame.

Since we have not implemented a scheduler for coroutines, we have taken our design choices for write/read semantics accordingly, so a program can run correctly. More specifically, a write to the input channel of the coroutine resumes coroutine up to the input channel read and then suspends its execution. On the other hand, reading from an output channel resumes coroutine up to the point it writes to the output channel, writes to channel, and then suspends its execution. Thus, after a completed communication of a channel control always is transferred to the caller function, whereas control could be transferred to the coroutine if the scheduler decided so. Either way, suspend and resume intrinsics allow the LLVM's pass manager to split the initial coroutine function to distinct parts of sequential codes (functions), which could be used by a runtime implementation for scheduling purposes. In Figure 5.5 we present transfer the control flow between a function and a coroutine, for each channel operation.

5.5 Compilation overview

In Section 5.3 we described how our code generator transforms a program's AST to LLVM IR. Then, the LLVM IR can be compiled to an object file, using the LLVM's static compiler (llc). The llc can be configured to compile for various *target triples*. Target triples in LLVM is a way to express in short the architecture, the Operating System and the abstract binary interface (ABI) of a compilation target. Thus, using the llc we can compile our Noisy programs to different object files. However, Noisy in its current state does not have a standard library implementation and thus cannot perform basic I/O operations, such as reading from stdin or writing to stdout.

In order to alleviate this problem, we created a small interface for Noisy to access the C libraries such as stdlib, stdio and math. Some functions of the C library can be linked without the need for an interface such as fabsf , or sin functions. However, functions like printf or scanf must be interfaced for each specific data type (or format) in the noisyLib, because Noisy's syntax in its current form does not provide a mechanism for declaring

functions with variable arguments and thus we could not create printf or scanf declarations from Noisy, and thus we are forced to create other functions to interface them like printInt32 or readInt32. The noisyLib is written in C and contains the function interfaces mainly for I/O operations. Also, since we cannot use the format specifiers, noisyLib contains functions for manipulating CSV files, which were used in our experiments. Finally, we can only perform I/O from standard input and output, because we have not yet implemented an interface for working with files.

Since noisyLib contains many function interfaces, the compilation process requires for the majority of our programs to be linked against it. Thus, our whole compilation process requires compiling noisyLib using a C compiler (both gcc and clang will do), and linking object files created by the Noisy compiler with the C library and noisyLib object file. In our experiments we used the ld linker. The whole compilation process is shown in Figure 5.6. We use the term *noisycc* to refer to the Noisy compiler.

Figure 5.5: *Coroutine control flow for different operations.*

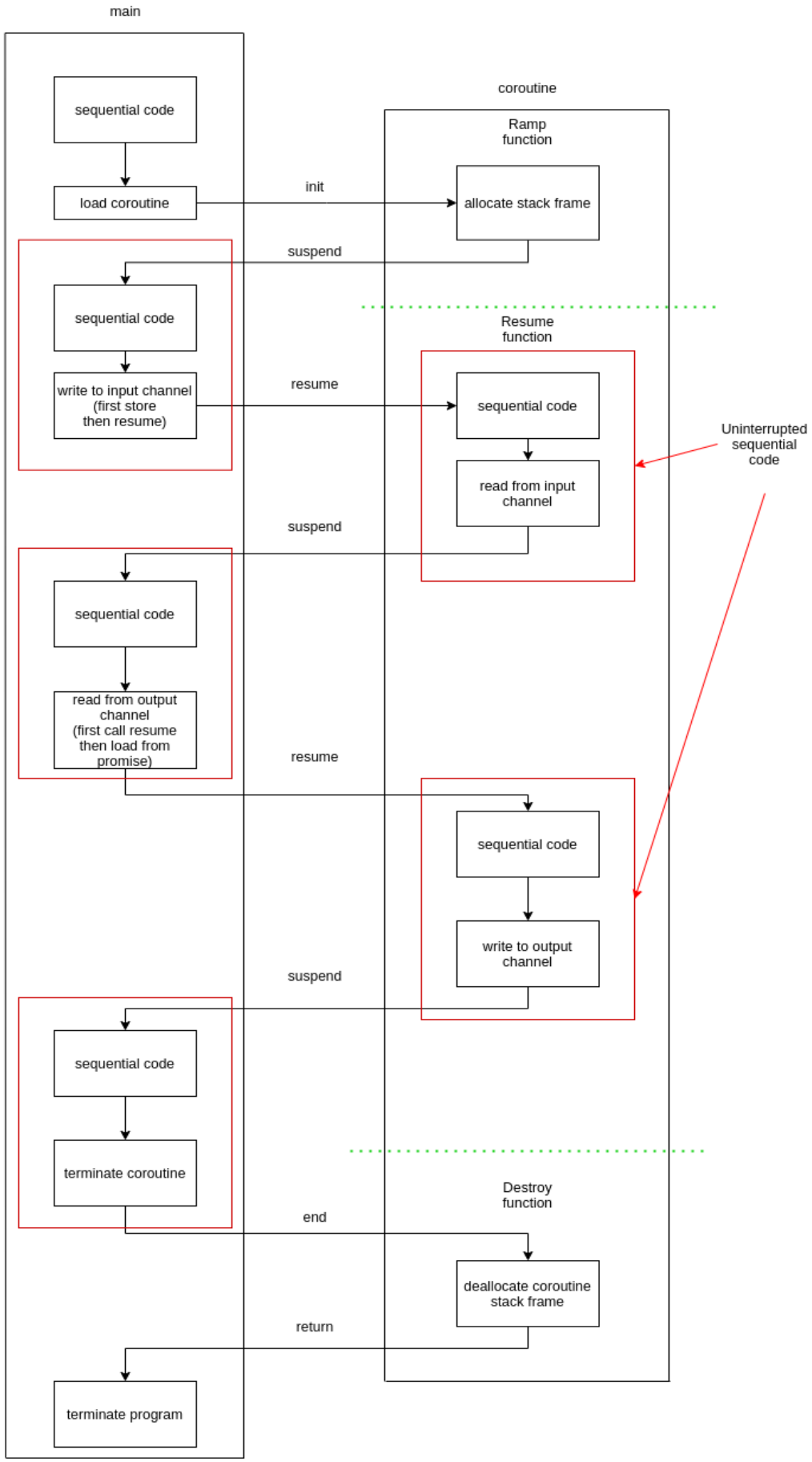
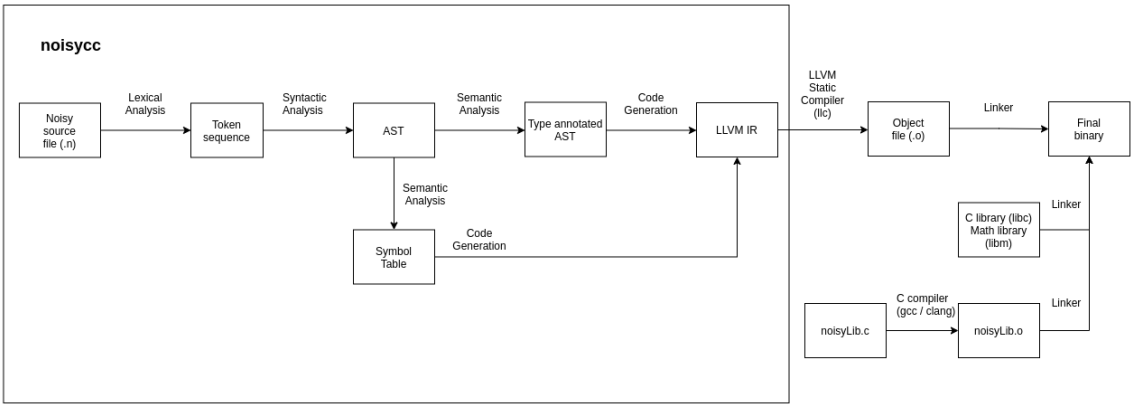


Figure 5.6: *Structure of the compilation process.*



Chapter 6

Evaluation

For the experimental evaluation of our work we implemented a variety of programs in Noisy and in C in order to compare their execution performance. Since we are using the LLVM infrastructure as our compiler backend and since LLVM based Clang compiler achieves similar results with gcc, we expect that our Noisy programs perform in a very similar way to C as well.

6.1 Applications

In this Section we will list the applications we decided to implement in order to test our compiler. These applications are mostly used in the embedded systems domain and showcase a number of different features in Noisy.

- **bme680**: This application consists of three different conversion routines that read temperature, pressure and humidity ADC values from the BME680 sensor and transform them to other integer values that can be used by an embedded system in order to further process them and produce results. The input of the program are values read from the sensor in a CSV file and the output of the application is the converted integer values. This application showcases the basic bit manipulation operations as well as type conversions between different arithmetic data types.

- **fib**: The fib application is a fibonacci number generator. In the source file we define a value N and the program prints the first N fibonacci numbers. The N value cannot be very large because the results of the program overflow the integer representations. This program showcases templated functions of Noisy as well as the use of coroutines. In the C implementation of fib, we try to emulate the coroutine implementation by using static variables.

- **lowPassFilter61**: The lowPassFilter61 is a low pass FIR filter application that uses 61 coefficient values. The Noisy implementation uses a coroutine to implement the filter. The input values of this program are measurements of a noisy sine wave and the output is the filtered values that plot a sine wave. We have a small and a larger input file.

- **lowPassFilter183**: The lowPassFilter183 is also an FIR filter but uses 183 coefficient instead. The implementation of this filter in Noisy is very similar with C. The input of this filter is the same with lowPassFilter61.
- **pedometer**: This application is a pedometer that receives as input values from acceleration sensors in a CSV file and prints the steps measured in its output.
- **pendulumEKF**: This application is an Extended Kalman Filter implementation for a pendulum's movement. The input of this application is noisy measurements of its speed and its output is the estimated position and speed of the pendulum. This application heavily uses matrix manipulation operations.
- **quicksort**: Quicksort is one of the most famous sorting algorithms. Its input is an array of 1000 elements and the output is the same array sorted.

Since Noisy does not have a standard library, all IO operations as well as complex math operations (e.g. absolute, sine etc.) are performed by the standard C library, for which we have created a small interface so Noisy programs can access it. Also, this interface forces us to use more IO calls than C would require, because Noisy does not support format specifiers, and as a result we also changed the C programs to perform the same IO calls as Noisy and use the same interface, in order to compare the two implementations more accurately.

6.2 Experimental Setup

6.2.1 Linux Experiments

For our experiments we ran the applications described in Section 6.1. The final binaries were produced using the GCC compiler for C and our Noisy compiler and LLVM infrastructure for Noisy, using three different optimization flags (-O0, -O1, -O2). The input of the programs is read from the standard input and their output is printed in the standard output. For their performance we evaluated two parameters; the static size of the final binary (after linking) and the dynamic instruction count for each execution. In order to measure the static size in bytes we used we used *size* tool [30] and in order to measure the number of dynamic instructions executed, we used *Intel's pin* tool [31]. The experiments were run on an Intel®Core™i7-9750H CPU @ 2.60GHz processor using an Ubuntu 20.04 OS. In Figures 6.1, 6.2, 6.3 we present the dynamic instruction count for the different programs and inputs. The name of the input is not mentioned in the graphs. In Figure 6.4 we compare the binary static size for all the programs and compiler flags.

In Table 6.1 we present the increase or decrease of the dynamic instruction count of the Noisy programs compared to their C counterpart as a percentage. We also present a mean value without the quicksort implementation because quicksort has a much worse performance than the rest of the programs for reasons we will explain in the next Section. In table 6.2 we present the increase or decrease of the static size of the final Noisy and C binaries as a percentage.

Figure 6.1: *Dynamic instruction count for fibonacci and BME680 conversion routines.*

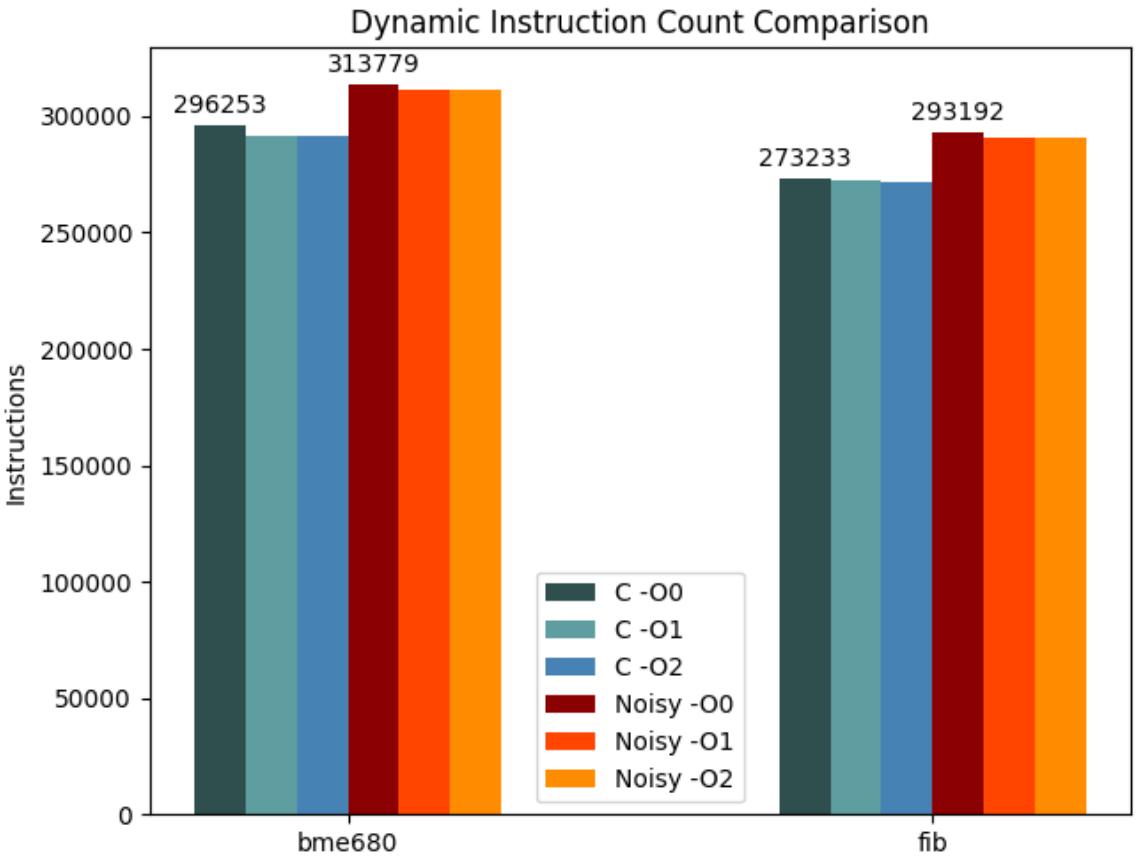


Figure 6.2: *Dynamic instruction count for low pass filters and quicksort.*

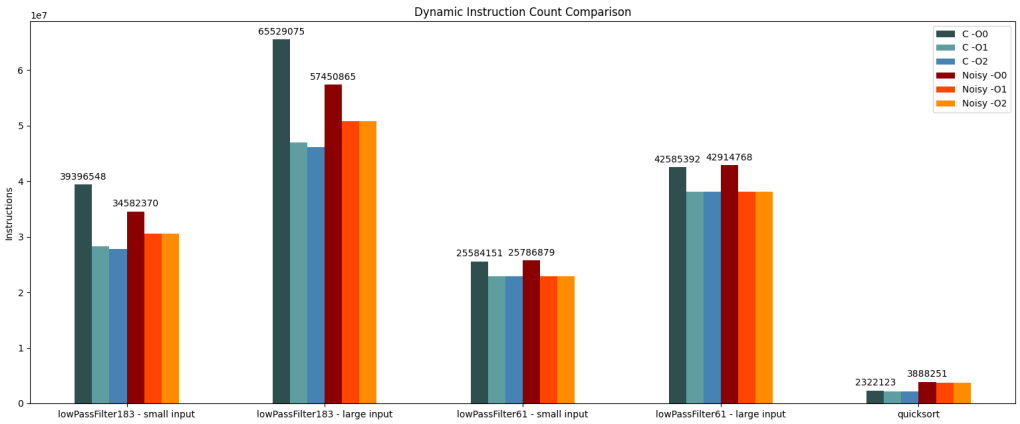


Figure 6.3: *Dynamic instruction count for the pedometer and EKF.*

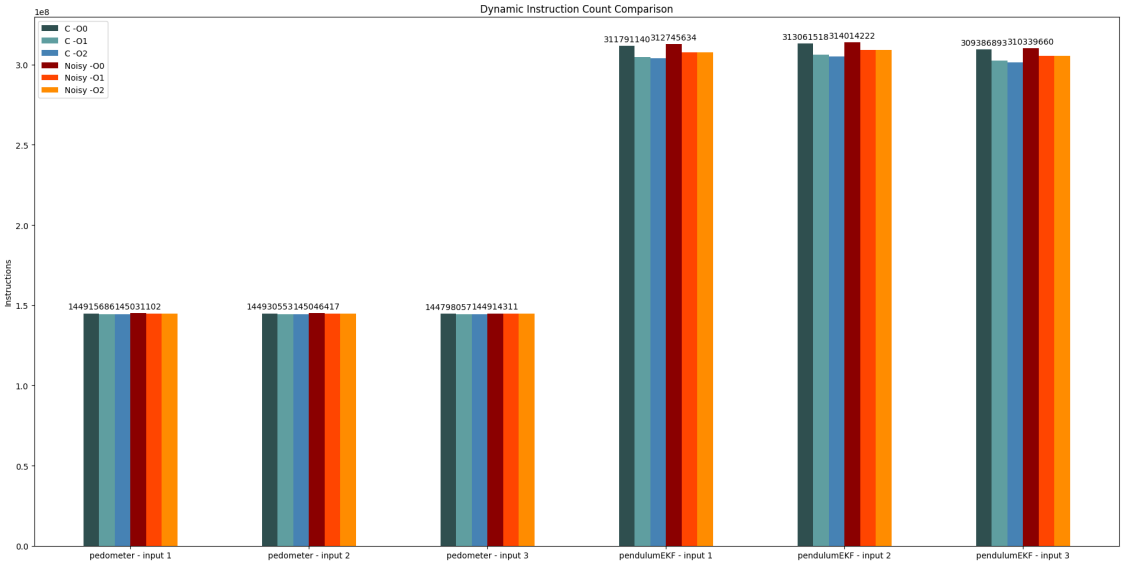
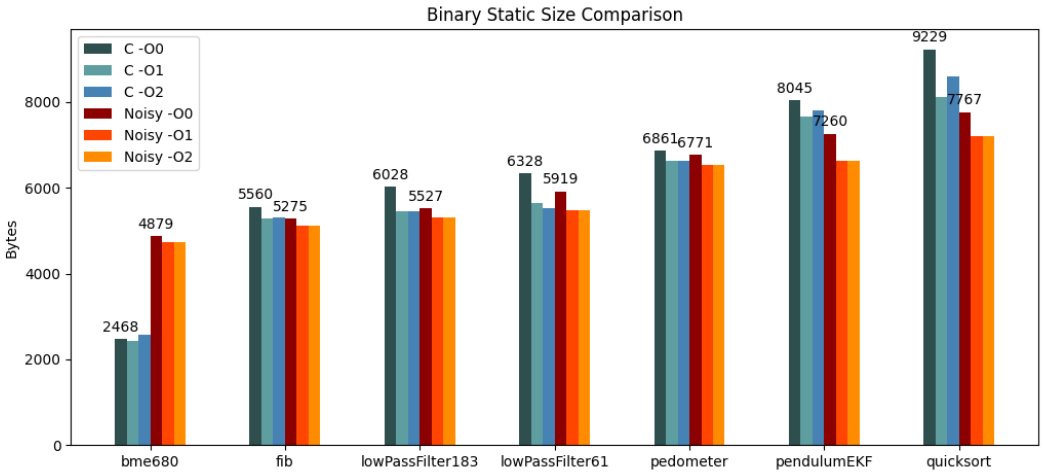


Figure 6.4: *Binary static size for all applications.*



Program	**-O0**	**-O1**	**-O2**
bme680	5.92%	6.85%	6.87%
fib	7.30%	6.63%	7.02%
lowPassFilter183 - all inputs	-12.27%	8.06%	10.05%
lowPassFilter61 - all inputs	0.79%	-0.16%	-0.16%
pedometer - all inputs	0.08%	0.20%	0.23%
pendulumEKF - all inputs	0.31%	0.96%	1.27%
quicksort	67.44%	76.41%	76.71%
Mean	11.6%	16.5%	17%
Mean w/o quicksort	0.43%	4.58%	5.06%

Table 6.1: *Dynamic Instruction comparison between C and Noisy programs.*

Program	-O0	-O1	-O2
bme680	97.69%	94.86%	83.38%
fib	-5.13%	-3.42%	-3.85%
lowPassFilter183	-8.31%	-2.51%	-2.51%
lowPassFilter61	-6.46%	-2.78%	-0.81%
pedometer	-1.31%	-1.22%	-1.22%
pendulumEKF	-9.76%	-13.50%	-15.05%
quicksort	-15.84%	-11.07%	-16.04%
Mean	7.27%	8.62%	6.27%
Mean w/o bme680	-7.80%	-5.75%	-6.58%

Table 6.2: *Binary static size comparison between C and Noisy programs.*

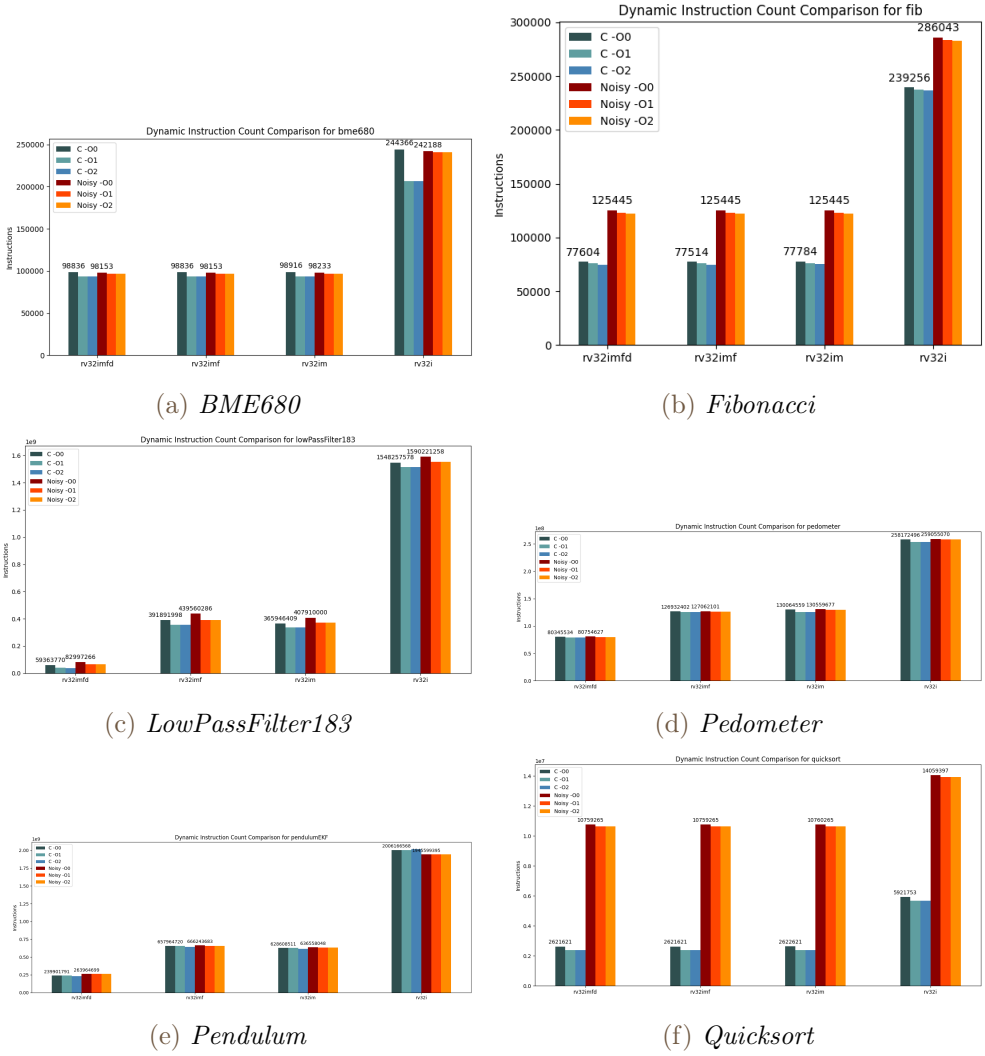
Based on the graphs we have produced, we measure that Noisy programs execute 5.06% more instructions than their C counter part, using the -O2 optimization flag which we consider to be the default for most compilers. In this mean value we have not included quicksort implementation which executes 76.71% more instructions, which is a pretty worse performance than the rest of the programs we implemented. For static size our programs are 6.58% smaller on average, excluding the bme680 implementation whose code produced is 83.38% larger.

6.2.2 RISC-V Experiments

For the second part of our experimental evaluation we tested the same applications by generating code for different RISC-V 32bit architectures [32], which are very common in the embedded systems domain. In order to compile our C programs we used the gcc to risc-v32 cross compiler and our Noisy compiler with the appropriate LLVM flags in order to compile Noisy to RISC-V. The code we generated was for 4 different architecture variants rv32imfd, rv32mfd, rv32im and rv32i. The rv32i architecture features only integer addition and subtraction, the rv32im adds integer multiplication and division, while in both architectures float numbers are software simulated. The rv32imfd and rv32imf feature hardware floating point numbers, with rv32imf to support single-precision floating point operations

and rv32imfd to also support double-precision floats. Our programs were compiled for the different architectures using three different optimization level flags (-O0, -O1, -O2).

For this experiment, we evaluated once again the static size of the final binaries and the dynamic instruction count for the programs execution. Unlike the previous experiment we chose only one input file for each program. In order to determine the static size we used again the *size* tool [30]. We also used the Sunflower embedded system simulator [8, 9], in order to run and measure the dynamic instruction count for our programs. In Figure 6.5 we present the dynamic instruction count for every program implementation for the different RISC-V 32-bit architecture extensions and in Tables 6.3, 6.4 we compare the dynamic instructions executed. Similarly, in Figure 6.6 we show the binary static size count of each program implementation and we compare the results in Tables 6.5, 6.6.

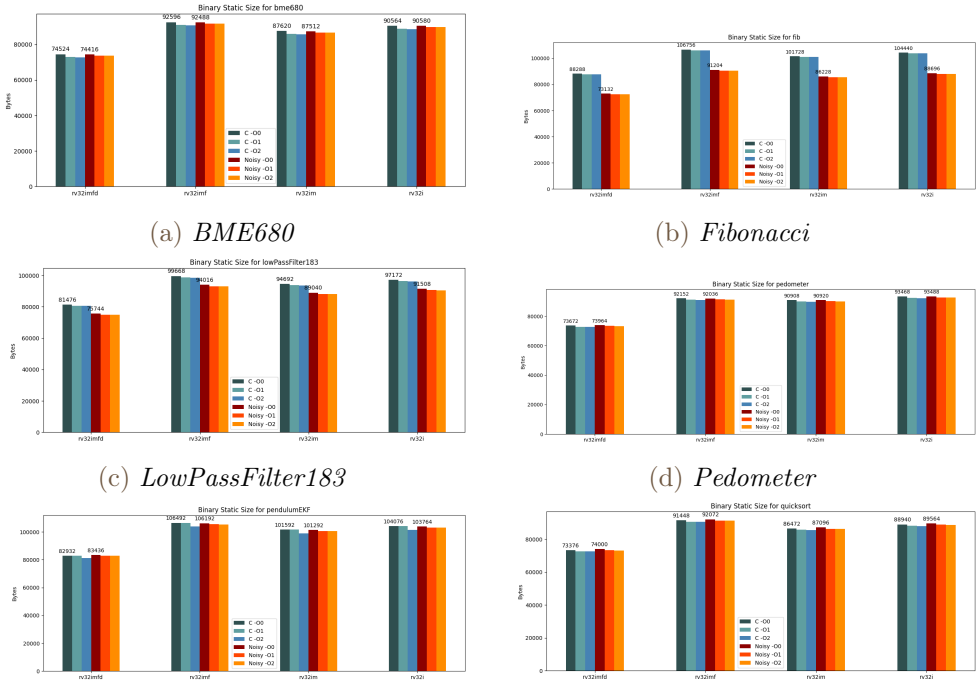


(a) *BME680* (b) *Fibonacci*

(c) *LowPassFilter183* (d) *Pedometer*

(e) *Pendulum* (f) *Quicksort*

Figure 6.5: *Dynamic instruction count for RISC-V 32-bit architecture.*



(a) *BME680* (b) *Fibonacci*

(c) *LowPassFilter183* (d) *Pedometer*

(e) *Pendulum* (f) *Quicksort*

Figure 6.6: *Binary static size for RISC-V 32-bit architecture.*

RISC-V Extension	IMFD			IMF		
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.69%	3.42%	3.44%	-0.69%	3.42%	3.44%
fib	61.65%	61.74%	63.46%	61.84%	61.94%	63.66%
lowPassFilter183	39.81%	64.01%	76.28%	12.16%	9.46%	9.75%
pedometer	0.51%	1.28%	1.27%	0.10%	0.58%	0.58%
pendulumEKF	10.03%	9.62%	13.58%	1.26%	-0.12%	2.52%
quicksort	310.41%	346.59%	348.34%	310.41%	346.59%	348.34%
Mean	70.29%	81.11%	84.40%	64.18%	70.31%	71.38%
Mean w/o quicksort	22.26%	28.02%	31.61%	14.93%	15.06%	15.99%

Table 6.3: *Dynamic Instruction comparison between C and Noisy programs for IMFD and IMF extensions of the RISC-V 32-bit architecture.*

RISC-V Extension	IM			I		
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.69%	3.42%	3.44%	-0.89%	16.63%	16.65%
fib	61.27%	61.36%	63.07%	19.56%	19.28%	19.63%
lowPassFilter183	11.47%	9.99%	10.30%	2.71%	2.47%	2.53%
pedometer	0.38%	3.60%	3.59%	0.34%	1.94%	1.94%
pendulumEKF	1.26%	0.97%	3.03%	-3.02%	-3.11%	-3.70%
quicksort	310.29%	346.45%	348.20%	137.42%	145.24%	145.60%
Mean	64.00%	70.97%	71.94%	26.02%	30.41%	30.44%
Mean w/o quicksort	14.74%	15.87%	16.69%	3.74%	7.44%	7.41%

Table 6.4: *Dynamic Instruction comparison between C and Noisy programs for IM and I extensions of the RISC-V 32-bit architecture.*

RISC-V Extension	IMFD			IMF		
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.14%	1.02%	1.12%	-0.12%	0.82%	0.90%
fib	-17.17%	-17.30%	-17.40%	-14.57%	-14.67%	-14.75%
lowPassFilter183	-7.04%	-7.22%	-7.10%	-14.57%	-14.67%	-14.75%
pedometer	0.40%	0.74%	0.86%	-0.13%	0.11%	0.23%
pendulumEKF	0.61%	-0.07%	2.06%	-0.28%	-0.94%	1.39%
quicksort	0.85%	0.91%	0.95%	0.68%	0.73%	0.76%
Mean	-3.75%	-3.65%	-3.25%	-3.35%	-3.29%	-2.86%

Table 6.5: *Binary static size comparison between C and Noisy programs for IMFD and IMF extensions of the RISC-V 32-bit architecture.*

RISC-V Extension	IM			I		
Program	-O0	-O1	-O2	-O0	-O1	-O2
bme680	-0.12%	0.86%	0.95%	0.02%	1.08%	1.14%
fib	-15.24%	-15.35%	-15.43%	-15.07%	-15.18%	-15.27%
lowPassFilter183	-5.97%	-6.11%	-6.01%	-5.83%	-5.97%	-5.87%
pedometer	0.01%	0.32%	0.40%	0.02%	0.36%	0.44%
pendulumEKF	-0.30%	-0.98%	1.46%	-0.30%	-0.97%	1.41%
quicksort	0.72%	0.78%	0.80%	0.70%	0.75%	0.78%
Mean	-3.48%	-3.41%	-2.97%	-3.41%	-3.32%	-2.89%

Table 6.6: *Binary static size comparison between C and Noisy programs for IM and I extensions of the RISC-V 32-bit architecture.*

6.3 Interpretation

In general, LLVM infrastructure is used to generate machine code really similar to C and also for the most features of Noisy we have chosen an implementation similar to that of C, except for arrays and coroutines. As a result, we expect our final binaries to have really similar performance with C. In both of our experimental setups we see that the dynamic

performance of Noisy programs is slightly worse than C, but close to it. For the static size of the programs, Noisy implementation is consistently smaller than their C counterpart. Based on the data we gathered, the array passing mechanism seems to create a substantial overhead when invoked many times in a program and we need to create an optimization pass to improve its performance or add more language features so the user can pass arrays by reference bypassing the copying mechanism, when needed. On the other hand, the coroutine feature, which is used for LowPassFilter61 and Fibonacci implementations, does not seem to add any substantial overhead and this is an encouraging result for a future development of the CSP model of the language.

The quicksort implementation is considerably worse in Noisy. Quicksort is a recursive algorithm and our current Noisy implementation uses *memcpy* in order to return or pass arguments of array type. As a result, in every recursive call of the quicksort function, memory is copied to return arrays and also the whole array is passed instead of a slice. Therefore, a lot of memory copying is happening in every function call, whereas in C it is not necessary, as C is using pointers and is accessing the array object directly instead of creating copies of it. This explains, the bad performance of our quicksort and it will be resolved by adding slices in the language and creating a new implementation.

Also, for the Linux experiment, the bme680 application in Noisy has almost double the static size of the C implementation. This is explained because we have not yet implemented structs in Noisy and we are forced to pass many function arguments in each bme680 function call instead of a struct that contains the same values. As a result the final code size is much bigger, which is not the case for the rest of the applications. The same issue is not so easily visible in the RISC-V experiment. To explain, for this experiment, most of a program's binary is the runtime libraries which are statically linked to the binary, since the application runs on a device without OS.

Chapter 7

Future Work

In Chapter 4 we presented the core ideas for Noisy language. In our work we implemented the core functionalities of the language for the basic arithmetic types and arrays, as well as a part of its CSP model, but we did not implement many of the original concepts of the language due to time constraints. These features could be implemented in a future work and can use the LLVM infrastructure and our implementation as a foundation to build upon and experiment further, to achieve the design goals for Noisy.

7.1 Type system

Noisy's type system features many ideas that can be experimented on, such as dimensional checking, units of measure, error tolerance. These concepts could be studied in terms of how they can be modeled inside a type system as well as how they can be checked efficiently. Also, Newton specification files could enrich the type system and add semantic information about signals, sensor and hardware descriptions that could be used by Noisy's type system. Also, there are many data types that have not been implemented yet, such as lists, tuples, abstract data types, complex numbers and dynamic data types. A future work could implement semantic check for these data types and also implement their representation in memory. Finally, one could experiment with compiler optimization passes that use information from the rich type system, regarding physical relations of values or restrictions of hardware.

7.2 Runtime system

In Section 5.4 we describe our implementation for the CSP model and why we believe the core features of this model can be implemented in a runtime system. The runtime system could implement channels, coroutine scheduling, M:N threading model to support parallelism for multi-core systems. Also, a runtime system could manage heap memory allocation for dynamic data types and lists. Moreover, the runtime system could have routines for runtime error handling, tolerance specification, bound check for arrays and other runtime functionalities. All these functionalities could be implemented as different subsystems of the runtime system and offer great opportunities for experimentation.

7.3 Uncertainty

An important aspect of Noisy's vision is to be able to express, model and handle uncertainty of values in programs. In (Bornholt et al., [33]) they show programming idioms to express uncertainty while in (Manousakis et al., [34]) they show methods to handle and propagate uncertainty through calculations. These methods can be used to express the uncertainty constructs in Noisy such as significant figures, or to support distribution values and variables that are featured in Newton as well. Finally, in (Tsoutsouras et al., [35]) they show a microarchitecture which can track the uncertainty on the hardware level. It would be really interesting for Noisy to be able to target this architecture and study how a language could benefit from hardware implementation.

7.4 Declarative Subset

Declarative subset of Noisy is responsible for proving properties of programs in Noisy and ensures correct execution of program, by identifying logical errors. The development of Noisy's declarative subset would require implementing semantic checking with Noisy programs and a backend that creates logical predicates based on the program variables. It is important that this feature is part of the language because the programmer does not need to use other tools and it should support integration with SMT solvers or other theorem provers.

Bibliography

[1] M. Barr, "Embedded systems glossary," *Neutrino Technical Library*, 2007.

[2] H. Steve, "Embedded systems design," *EDN Series for Design Engineers*, 2003.

[3] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[4] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[5] J. Lim and P. Stanley-Marbell, "Newton: A language for describing physics," *arXiv preprint arXiv:1811.04626*, 2018.

[6] "Llvm-c: C interface to llvm." https://llvm.org/doxygen/group__LLVMC.html. Date accessed: 11-2021.

[7] "Coroutines in llvm." https://llvm.org/docs/Coroutines.html. Date accessed: 11-2021.

[8] P. Stanley-Marbell and M. Hsiao, "Fast, flexible, cycle-accurate energy estimation," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, (New York, NY, USA), pp. 141–146, ACM, 2001.

[9] P. Stanley-Marbell and D. Marculescu, "Sunflower: Full-system, embedded, microarchitecture evaluation," in *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'07, (Berlin, Heidelberg), pp. 168–182, Springer-Verlag, 2007.

[10] W. Bich, M. G. Cox, R. Dybkaer, C. Elster, W. T. Estler, B. Hibbert, H. Imai, W. Kool, C. Michotte, L. Nielsen, *et al.*, "Revision of the 'guide to the expression of uncertainty in measurement'," *Metrologia*, vol. 49, no. 6, p. 702, 2012.

[11] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[12] M. Sipser, "Context-free grammars," *Introduction to the Theory of Computation*, pp. 91–122, 1997.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[14] W. M. Waite and G. Goos, *Compiler construction*. Springer Science & Business Media, 2012.

[15] B. Roscoe, "The theory and practice of concurrency," 1998.

[16] C. Hoare, "Communicating sequential processes prentice-hall," *Englewood Cliffs, NJ, USA*, 1985.

[17] W. D. Clinger, "Foundations of actor semantics," *AITR-633*, 1981.

[18] "Noisy language reference." https://github.com/phillipstanleymarbell/Noisy-lang-compiler/blob/master/docs/Noisy-language-reference.pdf. Date accessed: 11-2021.

[19] P. Winterbottom, "Alef language reference manual," *Plan 9 Programmer's Man*, 1995.

[20] D. M. Ritchie, "The limbo programming language," *Inferno Programmer's Manual*, vol. 2, 1997.

[21] A. A. Donovan and B. W. Kernighan, *The Go programming language*. Addison-Wesley Professional, 2015.

[22] Y. Wang, S. Willis, V. Tsoutsouras, and P. Stanley-Marbell, "Deriving equations from sensor data using dimensional function synthesis," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[23] V. Tsoutsouras, S. Willis, and P. Stanley-Marbell, "Deriving equations from sensor data using dimensional function synthesis," *Communications of the ACM*, vol. 64, no. 7, pp. 91–99, 2021.

[24] O. Kaparounakis, V. Tsoutsouras, D. Soudris, and P. Stanley-Marbell, "Automated physics-derived code generation for sensor fusion and state estimation," *arXiv preprint arXiv:2004.13873*, 2020.

[25] "Dot language." https://graphviz.org/doc/info/lang.html, Aug 2021. Date accessed: 11-2021.

[26] P. Stanley-Marbell, "Wirth-tools: Wirth is a set of tools to automate construction of a recursive-descent parser in the wirth style.." https://github.com/phillipstanleymarbell/Wirth-tools/. Date accessed: 11-2021.

[27] https://llvm.org/. Date accessed: 11-2021.

[28] "Rust by example." https://doc.rust-lang.org/rust-by-example/primitives/array.html. Date accessed: 11-2021.

[29] E. S. Pettersen, "Proxc–a csp-inspired concurrency library for the c programming language," *Department of Engineering Cybernetics, Norwegian University of Science and Technology*, 2016.

[30] "size(1)- linux man page." https://linux.die.net/man/1/size. Date accessed: 11-2021.

[31] "Pin - a dynamic binary instrumentation tool." https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. Date accessed: 11-2021.

[32] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, *et al.*, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[33] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain< t> a first-order type for uncertain data," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 51–66, 2014.

[34] I. Manousakis, Í. Goiri, R. Bianchini, S. Rigo, and T. D. Nguyen, "Uncertainty propagation in data processing systems," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 95–106, 2018.

[35] V. Tsoutsouras, O. Kaparounakis, B. Bilgin, C. Samarakoon, J. Meech, J. Heck, and P. Stanley-Marbell, "The laplace microarchitecture for tracking data uncertainty and its implementation in a risc-v processor," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1254–1269, 2021.