UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME GROCHAU AZZI

Semantics and Proof Calculus for Communicating Unstructured Code

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Trabalho realizado na Technische Universität Berlin dentro do acordo de dupla diplomação UFRGS - TU Berlin.

Orientador brasileiro: Prof. Dr. Rodrigo Machado
Orientadora alemã: Prof. Dr. Sabine Glesner
Co-orientador alemão: Dipl.-Math. Nils Jähnig

Porto Alegre
2015

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Azzi, Guilherme Grochau

Semantics and Proof Calculus for Communicating Unstructured Code / Guilherme Grochau Azzi. – Porto Alegre: CIC da UFRGS, 2015.

11 f.: il.

Trabalho de conclusão (graduação) – Universidade Federal do Rio Grande do Sul. Curso de Ciência da Computação, Porto Alegre, BR–RS, 2015. Orientador: Rodrigo Machado.

1. Linguagens não-estruturadas. 2. Semântica operacional. 3. Semântica axiomática. 4. Cálculo de processos. 5. Assistente de provas. I. Machado, Rodrigo. II. Título.

RESUMO

Software tende a se tornar cada vez mais onipresente e complexo. Seu uso em aplicações críticas, onde defeitos podem causar grandes danos materiais, ambientais ou até mortes, exige um alto grau de confiança em sua corretude. Para garantir tais níveis de confiança, a verificação formal é uma boa ferramenta, dado que pode *provar* propriedades do sistema ou de seus modelos. Métodos formais já são frequentemente empregados na verificação de hardware e software, sendo o software normalmente implementado ou modelado em linguagens com alto nível de abstração, cuja análise é mais simples. Tais linguagens são, no entanto, bastante diferentes do código de máquina que é efetivamente executado. A formalização de código não-estruturado pode, portanto, ser necessária para algumas aplicações, como a verificação de compiladores. Enquanto a verificação de programas sequenciais e estruturados já foi extensivamente explorada, programas concorrentes ou não-estruturados ainda são um problema em aberto. Além disso, a maioria das abordagens existentes lida apenas com um dos dois aspectos, e não com a sua combinação.

Neste trabalho é definida uma semântica formal para a linguagem Low-Level Do (LLDO), uma linguagem não-estruturada pequena e flexível que permite a especificação de processos comunicantes. A semântica operacional é apresentada tanto em estilo Small-Step quanto Big-Step. Além disso, um cálculo é proposto para a verificação de corretude de tais programas. Tanto a semântica Big-Step quanto o cálculo são definidos de maneira composicional ao combinar técnicas existentes que lidam individualmente com concorrência ou código não-estruturado. A aplicabilidade do cálculo é testada em alguns exemplos, e todo o trabalho é formalizado no assistente de provas Isabelle/HOL.

Palavras-chave: Linguagens não-estruturadas. semântica operacional. semântica axiomática. cálculo de processos. assistente de provas.

RESUMO ESTENDIDO

Este é um resumo estendido em português para a Universidade Federal do Rio Grande do Sul. O trabalho de conclusão original, em inglês, foi apresentado na Technische Universität Berlin através do programa de dupla diplomação UNIBRAL II entre as duas universidades.

1 Introdução

O Capítulo 1 introduz o tema geral deste trabalho, o tratamento formal de linguagens não-estruturadas ("*assembly-like*") para processos comunicantes. O objetivo é combinar técnicas existentes de forma a lidar de maneira composicional com ambas as características. Para isso, definem-se as semânticas operacional e axiomática da linguagem Low-Level Do (LLDo), uma linguagem simples e flexível.

2 Conceitos Básicos

O Capítulo 2 introduz brevemente os conceitos básicos empregados neste trabalho.

2.1 Semântica Formal

São apresentados os três principais estilos para a definição de semânticas formais. No estilo *operacional*, define-se uma máquina abstrata como modelo de execução da linguagem, geralmente através de um sistema de transição. Há duas variantes para esse estilo: na variante *small-step*, cada transição é uma ação atômica, utilizando-se o fecho transitivo do sistema para expressar execuções mais longas; na variante *big-step*, as transições representam execuções completas. O estilo *denotacional* promove o uso de objetos matemáticos para representar a semântica, utilizando uma função de interpretação que mapeia programas da linguagem para objetos de um domínio semântico. Tal estilo não é utilizado neste trabalho. O estilo *axiomático* define regras para analisar programas da linguagem, e seu uso como cálculo de inferência para demonstrar a corretude de programas é introduzido com mais detalhes.

2.2 Cálculo de Corretude

Uma maneira comum de especificar o comportamento de programas é através de pré- e pós-condições, sendo tal especificação satisfeita por um programa se sua execução, sempre que iniciada em estado que satisfaz as pré-condições, atinge apenas estados que satisfazem as pós-condições. Dado que a análise direta da semântica operacional ou denotacional do programa pode ser bastante trabalhosa, define-se um conjuto de regras de inferência para deduzir a corretudo de programas em relação a pré- e pós-condições — uma técnica proposta por (HOARE, 1969) que serve de base para muitas abordagens de verificação formal.

2.3 Eventos e Traços

Ao lidar com sistemas concorrentes, é necessário formalizar a comunicação entre processos. Para isso, a abordagem utilizada pelos diversos cálculos de processos — como CSP, CCS e cálculo pi — é tomada como base. Nestes cálculos, os processos se comunicam através de ocorrências sincronizadas de *eventos*. A execução de um sistema concorrente pode então ser entendida através de um *traço* dos eventos ocorridos. Uma técnica comumente empregada é o uso de eventos formados por pares (c, x) de um canal e um valor. Nesse caso, um dos processos geralmente aceita eventos de um canal com qualquer valor, enquanto outro processo aceita apenas um valor específico. Nesta situação, pode-se interpretar que o segundo processo envia um valor através do canal, enquanto o primeiro processo o recebe.

3 A Linguagem LLDo

O Capítulo 3 introduz a linguagem Low-Level Do, inicialmente expondo sua sintaxe e, a seguir, formalizando sua semântica no estilo operacional. A linguagem em si pode ser vista como um nível intermediário de abstração entre linguagens assembly, às quais um sistema é compilado, e modelos em cálculos de processo, utilizados para analisar tal sistema.

Como uma linguagem não estruturada, a interpretação de programas LLDo dependem do estado de uma *máquina abstrata*, que é composta por um banco de registradores — que mapeia cada registrador ao valor contido nele — e por uma tabela de processos — que mapeia cada identificador de processo a seu contador de programa, indicando a próxima instrução a ser executada neste processo. É importante ressaltar que processos indivudais têm uma vi-

são mais restrita da máquina, tendo acesso apenas a seu próprio contador de programa. Além disso, exige-se que processos operem apenas com subconjuntos disjuntos dos registradores da máquina, evitando a presença de estado compartilhado.

Em vez de compartilhar estado, os processos se comunicam através de eventos, dos quais são definidos três tipos. O *Evento vazio* τ representa a ausência de comunicação. Os atos de *envio* e *recebimento* de um valor v através de um canal c também são representados por eventos, denotados por $c!v$ e $c?v$ respectivamente — os dois eventos *não-sincronizados*. Já o evento de sincronização, denotado por $c.v$, sinaliza que dois processos já efetuaram a sincronização de envio e recebimento.

Os programas LLDo são definidos como conjuntos de processos (acompanhados de seus identificadores únicos), que por sua vez são conjuntos de instruções etiquetadas. Para possibilitar um tratamento composicional, porém, adota-se a abordagem de (SAABAS; UUSTALU, 2007): tais conjuntos são representados por árvores binárias cujas folhas contém os elementos do conjunto. Dessa forma, há uma estrutura a ser explorada de maneira composicional.

As instruções em si são definidas por cinco esquemas de instruções, que podem ser utilizados para representar boa parte das instruções comuns de linguagens assembly. Além disso, os esquemas são definidos em termos de lógica de alta ordem, utilizando por exemplo funções definidas na sintaxe da lógica. Os cinco esquemas são:

- `do`f modifica os registradores de acordo com a função f;
- `br` l modifica o fluxo de execução, prosseguindo com a instrução etiquetada por l;
- `cbr` $b\,l$ modifica o fluxo de execução condicionalmente, prosseguindo com a instrução etiquetada por l apenas quando o predicado b dos registradores é satisfeito;
- `send`$f\,c$ envia, através do canal c, um valor extraído dos registradores pela função f;
- `recv`$f\,c$ recebe, através do canal c, um valor que é utilizado para modificar os registradores de acordo com a função f.

3.1 Semântica Operacional

Na seção 3.4, a semântica operacional de LLDo é especificada. Duas variantes são propostas e demonstradas equivalentes: uma em estilo small-step, a outra em estilo big-step.

A variante small-step é definida de maneira não composicional, considerando os programas e processos como conjuntos e ignorando sua estrutura. Seu objetivo é expressar de forma intuitiva a semântica esperada. Assim, cada programa LLDo induz um sistema de transição

etiquetado cujos estados são os estados da máquina abstrata descrita anteriormente, e cujas etiquetas são os eventos causados pela execução de instruções. Cada transição causada por um programa representa ou a execução não-sincronizada de uma instrução por um único processo, ou a execução sincronizada de uma instrução por cada um de apenas dois processos. As relações de transição induzidas por processos e programas LLDo são definidas, respectivamente, nas figuras 3.8 e 3.9. Um lema importante sobre a a semântica small-step (Lema 3.2) é que as transições jamais modificam registradores que não são mencionados no programa.

Dado que a variante small-step não é composicional, uma variante big-step é definida utilizando as técnicas de Saabas and Uustalu (2007) e Hooman et al. (2003) para atingir a composicionalidade. A relação de transição big-step é definida nas figuras 3.12 e 3.13 para, respectivamente, processos e programas. De particular interesse são as regras de inferência BSSEQL e BSSEQR, que provêm a combinação sequencial de processos. Tais regras baseiam-se no fato de que a execução de um processo particionado em dois subconjuntos de instruções etiquetadas inicia sua execução em um desses subconjuntos. Após terminar a execução possível com as instruções do subconjunto inicial, porém, não basta executar apenas as instruções do segundo subconjunto — este pode desviar o fluxo de execução de volta ao primeiro. O restante da execução, portanto, deve considerar o processo inteiro.

Já a sincronização de processos é, na semântica big-step, baseada na sincronização de traços. Tal operação sobre dois traços, definida na figura 3.14, gera um conjunto de possíveis traços sincronizados, contendo em particular todas as intercalações dos traços. Além disso, todas as possíveis sincronizações de pares de eventos são contempladas. Como um exemplo, as possíveis sincronizações dos traços $\langle c!1, c!1\rangle$ e $\langle c?1\rangle$ são as seguintes.

$$\{\langle c.1, c!1\rangle, \langle c!1, c.1\rangle, \langle c!1, c!1, c?1\rangle, \langle c!1, c?1, c!1\rangle, \langle c?1, c!1, c!1\rangle\}$$

As execuções descritas pela semântica operacional podem conter, portanto, eventos não sincronizados. Dessa forma, pode-se recombinar a semântica de subprogramas para obter a semântica de um programa. Ao analisar um programa completo, porém, os eventos não-sincronizados são indesejados: eles indicam que o programa ficou parado esperando a sincronização, e que o resto do traço apenas seria alcançado se tal sincronização ocorresse. Nesse caso, devem-se considerar possíveis apenas as execuções cujos traços contém apenas eventos sincronizados.

Por fim, as duas variantes da semântica foram demonstradas equivalentes para execuções terminadas. Tal restrição é necessária pois a semântica big-step não expressa execuções parciais, nem execuções infinitas. Os passos mais importantes da demonstração podem ser expressos

como propriedades do fecho transitivo da semântica small-step. Para cada execução de um programa composto, por exemplo, existem execuções dos subprogramas cujos traços podem ser sincronizados de acordo com a execução original (Lema 3.11). Outro exemplo é que, para cada transição de um programa partindo de um estado s, se há um estado s' que difere de s apenas em registradores invisíveis ao programa, há também uma transição partindo de s' cujo estado final possui as mesmas diferenças invisíveis (Lema 3.10).

4 Cálculo de Corretude

O Capítulo 4 apresenta o cálculo de corretude, ou semântica axiomática, para a linguagem LLDo. Embora a semântica operacional forneça um entendimento rigoroso da linguagem, usá-la diretamente para verificar a corretude de programas é pouco prático. Uma abordagem para facilitar a verificação é baseada na semântica axiomática: especifica-se o programa através de uma pré-condição P e uma pós-condição Q, denotado por $\{P\}\ code\ \{Q\}$, e aplicam-se as regras de inferência da semântica para demonstrar que a especificação é satisfeita.

As asserções (pré- e pós-condições) são expressões lógicas que tratam do estado da máquina em um determinado momento, utilizando a variável h para se referir ao traço atual, σ para se referir ao banco de registradores, pc para se referir ao contador de programa (ao lidar com processos) e π para se referir à tabela de processos (ao lidar com programas).

A semântica axiomática é definida na Seção 4.1 e sua corretude quanto à semântica operacional big-step é demonstrada na Seção 4.2. As regras de inferência podem ser divididas em três grupos: regras para processos, regras para programas e regras lógicas ou genéricas. Os dois primeiros grupos são compostos por regras dirigidas pela sintaxe, que expressam a semântica da linguagem. Já o terceiro grupo é composto por regras independentes da sintaxe, expressando axiomas lógicos.

Apenas uma regra lógica é definida, HCONSEQ, e ela expressa a possibilidade de fortalecer pré-condições e enfraquecer pós-condições. As demais regras são baseadas na semântica operacional big-step. De particular interesse nas regras para processos são HDO e HRECV, em que o não-determinismo é modelado por um quantificador universal na pré-condição, além de HAPPEND que trabalha com uma invariante, dado que qualquer conjunto de instruções pode conter um laço. Nas regras para programas, HPAR deriva a pré-condição do programa composto a partir das pré-condições dos subprogramas individuais, afirmando que existem dois traços cuja sincronização pode resultar no traço composto, e que as pré-condições dos subprogramas são satisfeitas pelos respectivos traços individuais. A pós-condição é derivada da mesma forma.

Além disso, é exigido que as asserções dos subprogramas sejam restritas aos processos e registradores contidos neles, garantindo sua não-interferência.

5 Exemplos

O Capítulo 5 apresenta três exemplos de verificação de programas. Dado o carater genérico da linguagem LLDo, seu uso em exemplos concretos exige que suas definições sejam instanciadas. A Seção 5.1 provê tal instanciação, postulando que os registradores contém apenas números inteiros e definindo instruções concretas com base nos esquemas de instrução da linguagem.

A Seção 5.2 apresenta uma implementação da função de Fibonacci, demonstrando a verificação de processos sequenciais. De particular interesse na especificação do programa é a forma de expressar a não-modificação do traço: qualquer predicado satisfeito pelo traço inicial também deve ser satisfeito pelo traço final. Em particular, se para um traço t temos $h = t$ antes da execução, o mesmo deve valer após a execução.

Quanto à verificação de programas sequenciais, demonstra-se a divisão do programa em blocos que possuem um único ponto de entrada — sua primeira instrução — e um único ponto de saída — sua última —, mantendo um fluxo de execução linear. A verificação é feita primeiro para cada bloco individualmente, seguindo com a combinação dos blocos já verificados.

A Seção 5.3 formaliza parte de um padrão comum em sistemas distribuídos: “replicated workers”. Um processo “worker” é implementado, especificado e verificado independentemente da tarefa que ele executa — apenas a comunicação é abordada. Tal processo pode então ser instanciado para resolver tarefas arbitrárias. Em particular, a implementação de Fibonnaci pode ser utilizada e sua corretude já demonstrada é suficiente para a corretude do processo “worker”.

Por fim, a Seção 5.4 especifica um cliente simples que envia um único valor para um processo “worker”, seguido de um pedido de terminação. A corretude da sincronização de tal cliente com o processo “worker” definido na seção anterior é demonstrada.

6 Trabalhos Relacionados

O Capítulo 6 descreve trabalhos relacionados ao tema deste texto. Em particular, são mencionados trabalhos lidando com vários aspectos de linguagens não-estruturadas como pro-

priedades não-funcionais e modelos de memória detalhados.

7 Conclusão

Neste trabalho, a semântica da linguagem LLDo foi definida em estilos operacional e axiomático. A semântica operacional big-step foi definida de maneira composicional, bem como a semântica axiomática. Isso foi possível através da combinação de técnicas já existentes para código não-estruturado ou comunicante. A semântica axiomática foi utilizada para demonstrar a corretude de três pequenos exemplos.

Dado que a semântica axiomática definida neste trabalho lida apenas com execuções terminadas, ela não é apropriada para sistemas reativos. Uma extensão do cálculo introduzindo invariantes, como o utilizado por Hooman et al. (2003) para linguagens estruturadas, permitiria a verificação de tais sistemas. Para isso, porém, a semântica operacional big-step não seria apropriada.

Dado que propriedades não-funcionais (ex. temporais) também são importantes em aplicações de segurança crítica, esta é outra possível extensão. Em particular, o trabalho de Bartels and Glesner (2011) poderia ser tomado como base.

REFERÊNCIAS

BARTELS, B.; GLESNER, S. Verification of distributed embedded real-time systems and their low-level implementations using timed csp. In: IEEE. **Software Engineering Conference (APSEC), 2011 18th Asia Pacific**. [S.l.], 2011. p. 195–202.

HOARE, C. A. R. An axiomatic basis for computer programming. **Communications of the ACM**, ACM, v. 12, n. 10, p. 576–580, 1969.

HOOMAN, J. et al. **A Compositional Approach to Concurrency and its Applications**. [S.l.: s.n.], 2003.

SAABAS, A.; UUSTALU, T. A compositional natural semantics and hoare logic for low-level languages. **Theoretical Computer Science**, Elsevier, v. 373, n. 3, p. 273–302, 2007.

Fachgebiet Programmierung eingebetteter Systeme
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

Semantics and Proof Calculus for Communicating Unstructured Code

Bachelorarbeit

im Studiengang Informatik

Guilherme Grochau Azzi
Matr.-Nr. 354249

09.01.2015

Gutachter:
Prof. Dr. rer. nat. Sabine Glesner
Prof. Dr.-Ing. Uwe Nestmann

Betreuer:
Dipl.-Math. Nils Jähnig

Abstract

Software becomes ever more ubiquitous and complex. Its use in safety-critical environments, where errors may harm people or cost great amounts of money, requires a high level of confidence in its correctness. In order to ensure such levels of confidence, the use of formal verification methods to *prove* properties of the systems is an important tool. Such formal methods are widely applied to verify hardware and software, the latter usually written or modelled in high-level languages which are much easier to reason about. Such languages are, however, very different from the machine code which is actually executed. Therefore, the formalization of unstructured code may be necessary for some applications, such as the verification of compilers. Formal verification of sequential, structured programs has been extensively explored, but the techniques for concurrent or unstructured code are still an open problem. Furthermore, most approaches contemplate either concurrency and communication or unstructured code, not supporting the combination of the two.

In this thesis, operational semantics in small- and big-step style and a proof calculus are defined for Low-Level Do (LLDo), a small and flexible unstructured language for communicating processes which should be general enough to model more complex languages. The big-step semantics and the proof calculus are defined in a compositional way, unifying techniques already used for dealing with communicating or unstructured code individually. The applicability of the calculus is tested on some simple examples.

Zusammenfassung

Die Korrektheit von Software ist besonders bei sicherheitskritischen Anwendungen von entscheidender Bedeutung, wenn Fehler das Leben von Menschen gefährden können oder großen finanziellen Schaden anrichten. Um die Korrektheit sicherzustellen, sind formale Verifikationsverfahren wichtig, da sie Eigenschaften der Systeme mathematisch *beweisen* können. Für die Verifikation von Programmen, die in höheren, strukturierten Programmiersprachen geschrieben wurden, werden diese Verfahren schon häufig in der Praxis angewendet. Strukturierte Programmiersprachen unterscheiden sich allerdings sehr von unstrukturiertem Maschinencode, in den sie übersetzt werden und der anschließend tatsächlich durchgeführt wird. Die Formalisierung unstrukturierter Sprachen ist also in gewissen Fällen notwendig, beispielsweise für die Verifikation von Compilern. Obwohl einige Verfahren für die Verifikation von unstrukturiertem Code schon existieren und weitere für nebenläufige, kommunizierende Programme, können sie mit beiden Eigenschaften umgehen.

In dieser Arbeit wird eine operationale Semantik im Small-Step- und Big-Step-Stil sowie ein Korrektheitskalkül für Low-Level Do (LLDo) entwickelt. LLDo ist eine kleine, flexible und unstrukturierte Programmiersprache, die nebenläufige und kommunizierende Prozesse beschreibt. Um unstrukturierte und kommunizierende Programme kompositional behandeln zu können, werden existierende Ansätze für die jeweiligen Problemstellungen kombiniert. Die Anwendbarkeit des Kalküls wird mit einigen einfachen Beispielen gezeigt.

Acknowledgements

I would like to thank my advisor Nils Jähnig for always promptly and patiently helping whenever the need for guidance presented itself, and my supervisor Prof. Dr. Sabine Glesner for the lessons which motivated me to pursue the field of formal verification. I am also deeply thankful for the opportunities I've been given to study both at UFRGS[1] and at the TU Berlin. Without their inspiring and often challenging lessons, my life would be much duller. Prof. Ana Bazzan, I thank you for introducing me to research, taking me as a research assistant in my second semester at UFRGS. The experience I got by working with you goes much beyond the field of multiagent systems.

I also owe my deepest gratitude to my family and friends, which support and motivate me to keep studying and improving. I thank my father, who warned me from his own experience about the problems of a career in Computer Science and accepted my choice to follow it anyway. I thank my mother, who reminds me it is possible to get through every semester if I focus on one task at a time. I thank my brother, who often demonstrates that life must not just be about studying. I'm grateful for the always very informative conversations with Alexander Elvers, and for his help with translations into German. I'm also thankful to Ana Bordignon, to whom I can always turn when I just feel like complaining about the world. I still have many remarkable acknowledgements, which this page is too small to contain.

[1] Universidade Federal do Rio Grande do Sul

Contents

1 Introduction

Software grows ever more ubiquitous and complex. Its use in safety-critical environments, where errors may harm people or cost great amounts of money, requires a high level of confidence in its correctness. In order to ensure such levels of confidence, the use of formal verification methods to *prove* properties of the systems is an important tool.

Such formal methods are widely applied to verify hardware and software. The latter usually written or modelled in a high-level languages, which are much easier to reason about. They are, however, very different from the machine code which is actually executed. Therefore, the formalization of low-level languages may be necessary for some applications.

One such case is the verification of compilers, whose correctness is critical — their bugs may introduce errors to innumerable programs. To assert their correctness, however, we must compare the behaviour of the program in both source and target language, and therefore need a formal specification of such behaviours. Another application, Proof-Carrying Code, a technique for verifying the safety of imported modules proposed by Necula [11] that reconciles a high level of security with the efficiency of native machine code.

1.1 Problem

Formal verification of sequential, structured programs has been extensively explored. The techniques for concurrent or unstructured code are, however, still an open problem. Furthermore, most approaches contemplate either concurrency and communication or unstructured code, not supporting the combination of the two.

An important property of formalizations is compositionality, which is also harder to achieve in a concurrent or unstructured setting. Communication-based concurrent calculi such as CSP are compositional, but abstract and very far from low-level programming languages. Compositionality in unstructured code has also been achieved [14], but in the absence of concurrency.

1.2 Goals

The goal of this bachelor's thesis is to define operational semantics and a proof calculus for a small and flexible unstructured language for communicating processes — called Low-Level Do (LLDo) and defined in the VATES project at the Technis-

che Universität Berlin[2] —, which should be general enough to model more complex languages. The semantics shall be presented in small- and big-step style, being the latter compositional for easing the use of traditional verification techniques. The calculus shall also be compositional and focus on functional and partial correctness. Finally, the applicability of the proof calculus shall be tested on a few simple examples.

1.3 Approach

We define a non-compositional small-step operational semantics for LLDo, clearly reflecting the expected behaviour of its instructions and of the interactions between processes. A compositional semantics in big-step style and a proof calculus are also defined, unifying the approach of [14] for dealing with unstructured code and the approach of [7] for dealing with concurrency. The equivalence of the semantics in both styles and the correctness of the calculus with respect to them is proved in the Isabelle proof assistant. The application of the proof calculus is also performed in the proof assistant.

This thesis is organised as follows. Section 2 briefly describes the background necessary for understanding this work. Section 3 presents the Syntax and Semantics of LLDo, proving that the two presentations of the semantics are equivalent. Section 4 contains the definition of the Proof Calculus, a proof of its correctness regarding the previously defined semantics and simple examples of its use. Section 5 compares this thesis with related work, and section 6 presents the conclusions and possible future work.

[2] An early version of LLDo without support for communication may be found in [3].

2 Background

In this section, the necessary background for understanding this work is briefly described, referencing works that provide more in-depth explanations. Initially, the usual styles for presenting formal semantics are introduced — focusing on the operational one —, followed by the use of the proof calculi for verifying programs and by the notions of events and traces — used for dealing with communication.

2.1 Formal Semantics

In order to formally reason about the programs, we need a precise specification of their meaning or behaviour — a *formal semantics*. The formulation of such semantics is usually given in one of the operational, denotational or axiomatic styles.

"*Operational semantics* specifies the behaviour of a programming language by defining a simple abstract machine for it" [12]. This abstract machine, which uses the language itself as machine code, is usually given as a transition relation between states and often specified by inference rules used to prove that a transition is possible. There are also two styles for the presentation of operational semantics. In the *small-step* style, each transition represents an atomic computation step, and its transitive closure is used to achieve longer executions of a program. In the *big-step* style, each transition represents a terminating execution of the program. Pierce [12] provides a more in-depth introduction to operational semantics.

Denotational semantics is an alternative style, where each program is mapped by an *interpretation function* to a mathematical object of some *semantic domain*. Examples are the interpretation of expressions from the lambda calculus as mathematical functions, or of CSP-expressions as the sets of traces they may cause. This style will not be employed on this text.

Axiomatic semantics define the meaning of programs by providing laws for reasoning about them, such as proving that they satisfy a given specification. Although such laws are used in this thesis, they are not taken as a specification of the language, but rather as a tool for verifying programs. A more detailed explanation is provided in the next subsection.

2.2 Verification Calculus

The use of proof calculi for verifying the correctness of programs has been widely explored, and variations of Hoare's original approach [4] are still developed and applied. The approach consists of specifying the behaviour of a program with

pre- and postconditions. The specification is considered satisfied if, whenever started in a state that satisfies the preconditions, the execution of a program will always terminate in a state that satisfies the postconditions. A set of inference rules is thus provided for proving that programs satisfy their specification without explicitly reasoning about the all executions of a program.

2.3 Events and Traces

For a very long time, it has been necessary to reason about concurrent computation, that is, many computations that occur in overlapping time periods and may need to share resources or information. Two main approaches for concurrent programming have arisen: the explicit manipulation of shared resources and *event*-based or message-passing systems. While the former is closer to the way machines work, programs written in it are hard to reason about due to the high degree of interdependence between their concurrent parts.

A more abstract view is provided by event-based or message-passing systems. These include process calculi — such as CSP[5, 6], CCS[8] and the π-calculus[9] — and programming languages — such as Erlang and Go. In these systems, each sequential process of the program may pause execution and wait for an event to occur. Generally, an event is triggered when two processes are ready for it. By restricting the points in which the processes may interfere with each other, reasoning about these systems is simpler.

Events may be atomic or composite values, and a common pattern is the use of pairs (c, x) of a channel identifier and a transmitted value. In this case, one of the processes will generally wait for any event of a given channel, while another will synchronise only with a specific value. The latter process may then be seen as *sending* a value through the channel, with the former process receiving it. This is the idea of message-passing, as commonly implemented in programming languages.

When reasoning about the behaviours of programs, it is often necessary to specify the sequence of events that may happen during their execution. This is the notion of a *trace*: a list of events that happened during a particular execution of a program, in order of occurrence.

This thesis is mostly based on the CSP model of computation. In [13], its semantics are presented in various styles, each focusing the verification of different properties. These approaches are adapted in [7] for imperative programming languages, with a focus on the composability of the semantics.

3 The LLDo Language

This section introduces the syntax of the LLDo language and the states of its abstract machine. A description of their goals is also provided, as well as an informal introduction to their semantics. The formal semantics will be defined in the following sections.

The primary goal of the LLDo language is to help fill the gap between low-level implementations in assembly language and high-level models in calculi that emphasize communication, such as CSP. It should also be general enough to model many variants of assembly languages. Furthermore, the language should allow the definition of a compositional semantics.

The language itself is composed of three syntactic categories: programs, processes and instructions. Programs are collections of processes that run concurrently, processes are collections of labelled instructions and instructions are atomic units of computation. In order to enable compositional semantics, processes are not allowed to share state. They therefore operate on disjoint sets of registers, communicating only through channels.

LLDo is defined by a shallow embedding in Higher Order Logic. That is, instead of defining instructions that explicitly mention the registers and whose semantics are given by an interpretation function — e.g. an instruction $\texttt{add}\, r_d\, r_1\, r_2$, whose interpretation assigns to r_d the sum of the current values of r_1 and r_2 —, the instructions explicitly mention such an interpretation function — e.g. $\texttt{do}\, f$, where f modifies the machine state as intended.

This way, we only need five instruction schemes that are able to model most usual assembly instructions, greatly reducing the number of cases we need to analyse when reasoning about the language. On the other hand, reasoning about each instruction becomes more complex due to their generality. The application in concrete cases is also more complex, since the concrete instructions must be defined in terms of the general ones. Furthermore, properties such as which registers are affected by the instruction must be proven. Nevertheless, the gain in generality seems to outweigh the added complexity.

3.1 Machine States

The LLDo abstract machine contains two components: a register store and a process table. The register store maps each register to its value, while the process table maps each process identifier to its current Process Counter (PC), which indicates the next instruction to be executed. The machine is also polymorphic on the type of data that its registers store. This allows the modelling of many actual

machines, including those which have registers of varying types by using a disjoint union.

The LLDo machine doesn't strive to closely model existing machines, since they don't provide good primitives for communication and concurrency. Those tasks are usually fulfilled by operating systems. The machine model, therefore, models both machine and operating system.

The formal definition of a machine state in Isabelle may be seen in Figure 3.1. The register store is modelled by the type **regstore**, a function from registers to their contents. The record type **conc_state** models the complete state of the machine, as already described. This is the state as perceived by a concurrent program, composed of multiple processes. In order to reason about each sequential process independently, **seq_state** is used, containing the register store and a single PC, which corresponds to the process. This is the state as perceived by a single process. Both process identifiers (**pid**) and labels (**label**) are represented as natural numbers. In order to improve the readability of the text, `seq_state`s will often be written as pairs (σ, l) of the register store σ and PC value l, and `conc_state`s as pairs (σ, π) of register store σ and process table π.

A common operation is the "projection" of a concurrent state, to obtain the state as viewed by a particular process. This is the function **procState**, also defined in Figure 3.1.

type_synonym pid = nat
type_synonym label = nat

type_synonym 'val regstore = (reg ⇒ 'val)

record 'val seq_state =
  PC :: label
  R_s :: ('val regstore)

record 'val conc_state =
  PCs :: (pid ⇒ label)
  R_c :: ('val regstore)

$$\text{procState}((\sigma, \pi), p) \stackrel{def}{=} (\sigma, \pi(p))$$

Figure 3.1: Formal definition of machine states in Isabelle.

Although processes are not allowed to share registers, a global register store is used. This decision was taken to ease future work that does allow shared regis-

ters. In order to reason about concurrent processes we must now, however, prove non-interference. Therefore, a notion of *framed* differences is useful — i.e. the differences between two states or register stores are restricted to a given set of registers and possibly PIDs, that is, to a *frame*. This notion is formally defined in Figure 3.2. The predicate **framedChange** refers to two functions of the same domain, asserting that every member of their domain not contained in the frame is mapped by both of them to the same value. Such functions may represent, for example, process tables or register stores. The predicates $\mathbf{framedChange_s}$ and $\mathbf{framedChange_c}$ refer to states of sequential processes and concurrent programs, respectively. They both assert that the changes on register stores are framed, while the latter also asserts that the PCs of all processes not contained in the frame remain the same.

$$\frac{\forall\, x.\, x \notin A \longrightarrow f(x) = g(x)}{\text{framedChange}(f, g, A)} \qquad \frac{\text{framedChange}(\sigma, \sigma', R)}{\text{framedChange}_{\text{s}}((\sigma, l), (\sigma', l'), R)}$$

$$\frac{\text{framedChange}(\sigma, \sigma', R) \qquad \text{framedChange}(\pi, \pi', P)}{\text{framedChange}_{\text{c}}((\sigma, \pi), (\sigma', \pi'), R, P)}$$

Figure 3.2: Definition of framed changes.

Since we represent register stores and process tables as functions, we often need operators to modify functions. These are defined in Figure 3.3. We denote the modification of the value associated with a single member x of the domain by $f[x := z]$. The function update(f, A, g) simultaneously changes the values associated to all members of A, obtaining their new values from g. We use syntactic sugar to denote multiple, sequential modifications.

3.2 Channels and Events

The LLDo processes communicate through channels, of which there is an arbitrarily large set. The two possible communicating actions are sending a specific value or receiving any sent value. The use of channels is not restricted: every process may use any channel, and may both send and receive values through it. In order to reason about the communication, we use the notions of events and traces, formalized in Figure 3.4.

We consider four kinds of events. Internal events, denoted by $\boldsymbol{\tau}$, indicate that some internal change without communication has occurred. Sending and receiving

$$f[x := z](y) \stackrel{def}{=} \begin{cases} z & \text{if } y = x \\ f(y) & \text{if } y \neq x \end{cases}$$

$$f[x_1 := z_1, x_2 := z_2, ..., x_n := z_n] \stackrel{def}{=} f[x_1 := z_1][x_2 := z_2]...[x_n := z_n]$$

$$\text{update}(f, A, g)(x) \stackrel{def}{=} \begin{cases} g(x) & \text{if } x \in A \\ f(x) & \text{if } x \notin A \end{cases}$$

$$\text{update}(f, A_1, g_1, A_2, g_2, ..., A_n, g_n) \stackrel{def}{=} \text{update}(... \text{update}(\text{update}(f, A_1, g_1), A_2, g_2)..., A_n, g_n)$$

Figure 3.3: Operators used for modifying functions.

events — **evSend** and **evWait**, denoted on this text by $c!v$ and $c?v$, respectively — contain the channel c through which communication should occur and the value v that should be communicated. These two are the *unsynchronized* events (**unsynced$_e$**), since they indicate the need for synchronization. Finally, the synchronization events **evComm** — denoted in this text by $c.v$ — are analogous but indicate that both a sending and a receiving occurred.

```
type_synonym channel = nat

datatype 'val event =
  τ |
  evSend channel 'val (infix (!!) 65) |
  evWait channel 'val (infix (??) 65) |
  evComm channel 'val (infix (!?) 65)

primrec unsynced_e :: ('a event ⇒ bool) where
  (unsynced_e τ = False) |
  (unsynced_e (ch!!_) = True) |
  (unsynced_e (ch??_) = True) |
  (unsynced_e (ch!?_) = False)
```

Figure 3.4: Formalization of events and traces.

The behaviour through time of a process or program is described by a trace of such events. Traces arising from single processes can clearly only contain internal and unsynchronized events, since no communication can occur without a second process. Valid traces of complete programs, on the other hand, may not contain unsynchronized events — which is expressed by a predicate defined in Figure 3.11.

Furthermore, in order to reason about traces, we need a few more operators. The projection operator $t \restriction_{c_1 c_2 \ldots c_n}$ "filters" a trace, allowing only events that occurred on the specified channels. We also denote by length(t) the number of events that occurred in a trace; by $t[i]$, the i-th event of the trace (assuming $i <$ length(t)); and last(t) = t[length(t) − 1], assuming *length*(t) > 0. We also denote by val(e) the value contained in an event, assuming the event isn't τ.

3.3 Syntax and Informal Semantics

The language itself is composed of three syntactic categories: programs, processes and instructions. Programs are collections of processes that run concurrently, each of them associated with a unique process identifier (PID). Processes are collections of labelled instructions, which are supposed to be run in sequence until an undefined label is reached. There exists, therefore, a notion of "incrementing" a label, provided by its representation as a natural number. The instructions themselves are atomic units of computation. Although they explicitly mention labels and channels where needed, the handling of registers is defined by functions — written in the syntax of HOL — that directly manipulate the machine state as perceived by the sequential process.

The syntax of the language is shown in Figure 3.5. Both programs and processes — **conc_code** and **seq_code**, respectively — are defined as binary trees, an approach introduced by [14] that is key to the compositionality of semantics. Due to the shallow embedding, we may only define five **instruction** schemes, described in the next paragraphs.

The **do** f instructions are those that simply modify the values of registers, such as adding the value of two of them and storing the result in a third. The actual modification of the register store is defined by the function f. The fact that the result of applying f is a set of stores allows the expression of failure — by resulting in an empty set — and non-determinism when needed.

Two usual branch instructions are also provided. The unconditional branch **br** proceeds the execution from the given label, instead of from the successor of the current PC. The conditional branch **cbr** b only branches when the predicate b holds for the current register store.

In order to provide communication between processes, two communicating instructions are provided — **send** and **recv**. Both communicate through a single specified channel. In `send` f c, the sent value is determined by a the function f of the current register store. In `recv` f c, the modified register store is determined by the function f of the original store and of the received value.

In order to enforce the restriction that no register is used by more than a single

datatype 'val instruction =
  do ('val comm) |
  br label |
  cbr ('val bexp) label |
  recv ('val ⇒ 'val regstore ⇒ 'val regstore) channel |
  send ('val regstore ⇒ 'val) channel

datatype 'val seq_code =
  none ((∅)) |
  one label ('val instruction) (**infix** ($::_s$) 53) |
  append ('val seq_code) ('val seq_code) (**infixr** (⊕) 52)

datatype 'val conc_code =
  sequential pid ('val seq_code) (**infix** ($::_c$) 51) |
  parallel ('val conc_code) ('val conc_code) (**infixr** (‖) 50)

Figure 3.5: Syntax of LLDo as defined in Isabelle.

process, we need to be able to retrieve the set of registers that affect or are affected by a process or instruction. While the set of registers used by a process is clearly the union of the sets obtain from each of its instructions, the case of the instructions is complicated by their generality. The simple notion to define is however that of registers not being used. Concretely, the predicate **notUsed**(i, P), defined in Figure 3.6, asserts that no element of the set P of registers is used by the instruction i.

Registers are not used by `do` or `recv` when the modification of their values may occur before or after executing the instruction, leading to the same results (NUDO, NURECV). A register is not used by `cbr` when changing it or not doesn't affect the validity of the predicate (NUCBR). Analogously, a register is not used by `send` when the sent value is independent of changes to the register (NUSEND). The instruction `br` uses no registers (NURECV).

The set $\mathbf{regs_i}(i)$ of registers that affect or are affected by the instruction i is the least set such that all its members are not used by i.

Given the generality of the instructions, we can't define a procedure for obtaining the registers used by any arbitrary instruction. It is, however, easy to define trivial procedures for the instructions used in practice, proving they satisfy the definition. A simple example is the instruction for adding the values of two registers and

storing the result in a third:

$$\texttt{add}\, r_d\, r_1\, r_2 \stackrel{def}{=} \texttt{do}\, (\lambda\, \sigma.\ \sigma[r_d := \sigma(r_1) + \sigma(r_2)])$$
$$\mathrm{regs_i}(\texttt{add}\, r_d\, r_1\, r_2) = \{r_d, r_1, r_2\}$$

$$\frac{\forall\, \sigma\, \rho. f\, (\mathrm{update}(\sigma, R, \rho)) = \{\mathrm{update}(\sigma', R, \rho) \mid \sigma' \in f\, (\sigma)\}}{\mathrm{notUsed}\, (\texttt{do}\, f, R)}\ \textsc{NUdo}$$

$$\frac{}{\mathrm{notUsed}\, (\texttt{br}\, l, R)}\ \textsc{NUbr} \qquad \frac{\forall\, \sigma\, \rho.\ b(\sigma) \longleftrightarrow b(\mathrm{update}(\sigma, R, \rho))}{\mathrm{notUsed}\, (\texttt{cbr}\, b\, l, R)}\ \textsc{NUcbr}$$

$$\frac{\forall\, \sigma\, \rho\, x.\ \mathrm{update}(f(x, \sigma), R, \rho)) = f(x, \mathrm{update}(\sigma, R, \rho))}{\mathrm{notUsed}\, (\texttt{recv}\, f\ c, R)}\ \textsc{NUrecv}$$

$$\frac{\forall\, \sigma\, \rho\, x. f(\sigma) = f(\mathrm{update}(\sigma, R, \rho))}{\mathrm{notUsed}\, (\texttt{send}\, f\ c, R)}\ \textsc{NUsend}$$

$$\mathrm{mayBeUsed}\, (\mathit{instr}, R) \stackrel{def}{\iff} \forall\, S.\, S \cap R \longrightarrow \mathrm{notUsed}\, (\mathit{instr}, S)$$
$$\mathrm{regs_i}(\mathit{instr}) = R \stackrel{def}{\iff} \mathrm{mayBeUsed}\, (\mathit{instr}, R) \wedge (\forall\, S.\, \mathrm{mayBeUsed}\, (\mathit{instr}, S) \longrightarrow R \subseteq S)$$

$$\mathrm{regs_s}(\mathit{proc}) \stackrel{def}{=} \bigcup_{(l, \mathit{instr}) \in \mathit{proc}} \mathrm{regs_i}(\mathit{instr})$$
$$\mathrm{regs_c}(\mathit{prog}) \stackrel{def}{=} \bigcup_{(p, \mathit{proc}) \in \mathit{prog}} \mathrm{regs_s}(\mathit{proc})$$

Figure 3.6: Sets of registers used by an instruction, program or process.

A few well-formedness properties, defined in Figure 3.7 are needed in order for the definitions above to make sense. Well-formed instructions must use a defined set of registers. Well-formed processes are composed only of well-formed instructions and may only define a label once — this is ensured by demanding that, in any concatenation of processes, the sets labels defined by each subprocess are disjoint. In well-formed programs, all processes are well-formed and each register is used by at most one process — which is ensured by demanding that, in any parallel composition, the sets of used registers of the subprograms are disjoint.

$$\begin{aligned}
\text{wellFormed}_\text{i}(\mathit{instr}) &\overset{def}{\iff} \exists\, R.\, R = \text{regs}_\text{i}(\mathit{instr}) \\
\\
\text{wellFormed}_\text{s}((l, \mathit{instr})) &\overset{def}{\iff} \text{wellFormed}_\text{i}(\mathit{instr}) \\
\text{wellFormed}_\text{s}(\mathit{proc}_1 \oplus \mathit{proc}_2) &\overset{def}{\iff} \text{wellFormed}_\text{s}(\mathit{proc}_1) \wedge \text{wellFormed}_\text{s}(\mathit{proc}_2) \wedge \\
&\qquad \text{dom}(\mathit{proc}_1) \cap \text{dom}(\mathit{proc}_2) = \emptyset \\
\\
\text{wellFormed}_\text{c}((p, \mathit{proc})) &\overset{def}{\iff} \text{wellFormed}_\text{s}(\mathit{proc}) \\
\text{wellFormed}_\text{c}(\mathit{prog}_1 \parallel \mathit{prog}_2) &\overset{def}{\iff} \text{wellFormed}_\text{c}(\mathit{prog}_1) \wedge \text{wellFormed}_\text{c}(\mathit{prog}_2) \wedge \\
&\qquad \text{regs}_\text{c}(\mathit{proc}_1) \cap \text{regs}_\text{c}(\mathit{proc}_2) = \emptyset \wedge \\
&\qquad \text{pids}(\mathit{proc}_1) \cap \text{pids}(\mathit{proc}_2) = \emptyset
\end{aligned}$$

Figure 3.7: Well-formedness predicates.

3.4 Operational Semantics

In order to formally reason about the LLDo language, a formal semantics for it is required. The most intuitive style for presenting it is arguably in the small-step operational style, which clearly describes the behaviour of each instruction and the interleaving and synchronisation of processes. Compositionality is, however, not ensured. Therefore, we also define a compositional semantics in big-step style, which is also more appropriate for reasoning about the verification calculus.

3.4.1 Small-Step Semantics

The main goal of the small-step semantics defined in this section is providing an intuitive specification for the language, not necessarily compositional. Therefore, the transition relations ignore the structure of processes and programs, treating them as sets of labelled instructions or processes.

Formally, the semantics of a process or program are defined by a ternary transition relation of the original state, the event associated to the transition and the state following the transition. It may therefore be seen as a labelled transition system, where events are the labels. We denote a transition of program or process p from

state s to state s', producing event e, by:

$$s \xrightarrow[p]{e} s'$$

The transition relation for a single process, defined in Figure 3.8, depends mainly on the current instruction to be executed — the instruction that is associated to the PC of the initial state. Three of the instructions — `do`, `br` and `cbr` — require no communication and are therefore always labelled by τ.

The rule SSDO describes all possible transitions for a $\texttt{do}\, f$ instruction — which all increment the PC, having as final register store a member of the set $f(\sigma)$, where σ is the original register store. Unconditional branches are defined by the rule SSBR and have a single possible transition: the PC is changed to the label indicated by the instruction. Conditional branches have two possible transitions — SSCBR_T and SSCBR_F. If the predicate is true for the original register store, the instruction behaves like the unconditional branches; if the predicate is false, the PC is simply incremented. In any case, all branch instructions leave the register store unchanged.

Both communicating instructions transition into a state where the PC is incremented. Sending instructions (SSSEND) will not modify the register store, and are labelled by the sending event of a specific value — obtained from the original register store by the specified function — through the specified channel. Receiving instructions (SSRECV), on the other hand, may be labelled with the receive event of any value through the specified channel. The resulting register store is then the result of applying the specified function to the original store and to the received value.

$$\frac{(l, \texttt{do}\, f) \in \mathit{proc} \qquad \sigma' \in f(\sigma)}{(\sigma, l) \xrightarrow[\mathit{proc}]{\tau} (\sigma', \mathrm{succ}\, l)}\ \textsc{SSDO} \qquad \frac{(l, \texttt{br}\ l') \in \mathit{proc}}{(\sigma, l) \xrightarrow[\mathit{proc}]{\tau} (\sigma, l')}\ \textsc{SSBR}$$

$$\frac{(l, \texttt{cbr}\ b\ l') \in \mathit{proc} \qquad b(\sigma)}{(\sigma, l) \xrightarrow[\mathit{proc}]{\tau} (\sigma, l')}\ \textsc{SSCBR\_T} \qquad \frac{(l, \texttt{cbr}\ b\ l') \in \mathit{proc} \qquad \neg b(\sigma)}{(\sigma, l) \xrightarrow[\mathit{proc}]{\tau} (\sigma, \mathrm{succ}\, l)}\ \textsc{SSCBR\_F}$$

$$\frac{(l, \texttt{recv}\, f\ c) \in \mathit{proc}}{(\sigma, l) \xrightarrow[\mathit{proc}]{c?x} (f(x, \sigma), \mathrm{succ}\, l)}\ \textsc{SSRECV} \qquad \frac{(l, \texttt{send}\, f\ c) \in \mathit{proc}}{(\sigma, l) \xrightarrow[\mathit{proc}]{c!f(\sigma)} (\sigma, \mathrm{succ}\, l)}\ \textsc{SSSEND}$$

Figure 3.8: Small-Step transition relation for sequential processes.

An important property of the small-step transition relation is that the only registers modified by it are those in the set of registers of the process. This is expressed by the following lemma, which was formalized in Isabelle.

Lemma 3.1. $s \xrightarrow[proc]{t} s' \Longrightarrow \text{framedChange}_\text{s}(s, s', \text{regs}_\text{s}(proc))$

Proof. Let $s = (\sigma, l)$ and $s' = (\sigma', l')$. By the definition of framing, we need to prove the following fact.

$$r \notin \text{regs}_\text{s}(proc) \Longrightarrow \sigma(r) = \sigma'(r)$$

We therefore assume $r \notin \text{regs}_\text{s}(proc)$ and proceed by induction on the derivation of the transition. We only prove the case for SSDO. The other cases are analogous.

Assume $(l, \texttt{do}\, f) \in proc$ and $\sigma' \in f(\sigma)$. Since r is not in the set of registers used by the program, it must also hold that:

$$\begin{aligned}
& r \notin \text{regs}_\text{i}(\texttt{do}\, f) \\
\Longrightarrow\ & \text{notUsed}\,(\texttt{do}\, f, \{r\}) \\
\Longrightarrow\ & \forall\, \rho.\, f(\text{update}(\sigma, \{r\}, \rho)) = \{\text{update}(\sigma', \{r\}, \rho) \mid \sigma' \in f(\sigma)\} \\
\Longrightarrow\ & f(\text{update}(\sigma, \{r\}, \sigma)) = \{\text{update}(\sigma', \{r\}, \sigma) \mid \sigma' \in f(\sigma)\} \\
\Longrightarrow\ & f(\sigma) = \{\text{update}(\sigma', \{r\}, \sigma) \mid \sigma' \in f(\sigma)\} \\
\Longrightarrow\ & \sigma' \in \{\text{update}(\sigma', \{r\}, \sigma) \mid \sigma' \in f(\sigma)\}
\end{aligned}$$

Clearly, all members of the set $\{\text{update}(\sigma', \{r\}, \sigma) \mid \sigma' \in f(\sigma)\}$ must have the same value on register r as σ. In particular, this holds for σ'.

□

The semantics of programs, presented in Figure 3.9, are defined abstractly in terms of the transitions of individual processes. There are, then, two possible types of transition: the transition of a single process may be promoted to a transition of the whole program (SSONE), or two processes may synchronize and transition simultaneously (SSSYNC).

Since processes and programs have different views into the state of the machine, care must be taken when integrating the two. The PCs are easy to handle: the PC viewed from a process is simply the entry from the process table that corresponds to its PID. The register stores, on the other hand, must be implicitly partitioned into the regions — or *frames* — that are accessible by each process. Since we require that no register is used by more than one process, these frames are disjoint, and we therefore need only assert that the changes causes by each process are confined

$$\frac{(p, proc) \in prog \qquad (\sigma, \pi(p)) \xrightarrow[proc]{e} (\sigma', l)}{(\sigma, \pi) \xrightarrow[prog]{e} (\sigma', \pi[p := l])}\ \textsc{SSone}$$

$$\frac{\begin{array}{c}(p_1, proc_1) \in prog \qquad (p_2, proc_2) \in prog \\ (\sigma, \pi(p_1)) \xrightarrow[proc_1]{c!x} (\sigma_1, l_1) \qquad (\sigma, \pi(p_2)) \xrightarrow[proc_2]{c?x} (\sigma_2, l_2) \\ p_1 \neq p_2 \qquad \mathrm{regs_i}(proc_1) \cap \mathrm{regs_i}(proc_2) = \emptyset \\ \sigma' = \mathrm{update}(\sigma, \mathrm{regs_s}(proc_1), \sigma_1, \mathrm{regs_s}(proc_2), \sigma_2)\end{array}}{(\sigma, \pi) \xrightarrow[prog]{c.x} (\sigma', \pi[p_1 := l_1, p_2 := l_2])}\ \textsc{SSsync}$$

Figure 3.9: Small-Step transition relation for concurrent programs.

to their frames and only modify the original store within the frames of processes that transitioned.

Analogously to the transitions of sequential processes, small-step transitions of concurrent programs are also framed. The following lemma was also proved in Isabelle.

Lemma 3.2. $s \xrightarrow[prog]{t} s' \Longrightarrow \mathrm{framedChange_c}(s, s', \mathrm{pids}(prog), \mathrm{regs_c}(prog))$

Proof. We prove by induction on the derivation of the transition. We only prove the case for SSsync. The other case is very similar.

In this case, we have the following assumptions:

$$\begin{array}{c}(p_1, proc_1), (p_2, proc_2) \in prog \\ (\sigma, \pi(p_1)) \xrightarrow[proc_1]{c!x} (\sigma_1, l_1) \ \wedge \ (\sigma, \pi(p_2)) \xrightarrow[proc_2]{c?x} (\sigma_2, l_2) \\ \sigma' = \mathrm{update}(\sigma, \mathrm{regs_s}(proc_1), \sigma_1, \mathrm{regs_s}(proc_2), \sigma_2) \\ \pi' = \pi[p_1 := l_1, p_2 := l_2]\end{array}$$

The changes caused by the transition are all caused by either of the sub-transitions, which are framed to the registers of their respective processes. Since these are all contained in the registers of the program, all the changes must also be framed by the registers of the program.

Furthermore, the only changes from π to π' are on registers p_1 and p_2. Since they are both members of $\mathrm{pids}(prog)$, the framing of the process table also holds.

□

$$\frac{}{s \xrightarrow[prog]{\langle\rangle, 0}{}^{*}\ t}\ \textsc{MSZero} \qquad \frac{s \xrightarrow[prog]{e} s' \qquad s' \xrightarrow[prog]{es, k}{}^{*}\ t}{s \xrightarrow[prog]{e \frown es, k+1}{}^{*}\ t}\ \textsc{MSStep}$$

Figure 3.10: Step-indexed multi-step transition relation. The non-step-indexed version is defined analogously.

In order to reason about complete executions of processes and programs, we use a multi-step transition relation — essentially a reflexive and transitive closure of the transition relations, where the events are accumulated into traces — and its step-indexed variant, the latter defined in Figure 3.10. A multi-step transition of program or process p from state s to s', producing the trace t in i steps, is denoted in step-indexed and otherwise form as follows:

$$s \xrightarrow[p]{t, i}{}^{*}\ s' \qquad\qquad s \xrightarrow[p]{t}{}^{*}\ s'$$

Not all multi-step transitions are valid executions of a program, since they neither ensure that termination was reached, nor that all necessary synchronizations occurred. This presence of "incomplete" executions will be necessary when comparing the small-step and big-step semantics.

Since a notion of terminating execution is still needed, we define it in Figure 3.11 as a two predicates. States of terminating executions are stuck, which means that its PC is not in the domain of the process. In the case of concurrent programs, the PCs of all its processes are not in their respective domains. In terms of traces, they must be fullySynced, which means they doesn't contain unsynchronized events.

3.4.2 Big-Step Semantics

Since the small-step semantics is not compositional, we provide an alternative definition in big-step style. This is defined as an inductive relation of the program or process, the original state, the trace associated to the transition and the final state. We denote the execution of program or process p from state s to state s', producing trace t, by:

$$s \xrightarrow[p]{t} s'$$

The big-step relation for a single process is defined in figure 3.12. Since a process can only act when the PC is defined by it, the rule BSNop provides the non-

$$\begin{aligned}
\text{stuck}_\text{s}((\sigma, l),\ proc) &\overset{def}{\iff} l \notin \text{dom}(proc) \\
\text{stuck}_\text{c}((\sigma, \pi),\ es,\ prog) &\overset{def}{\iff} \forall (p, proc) \in prog.\ \pi(p) \notin \text{dom}(proc)
\end{aligned}$$

$$\begin{aligned}
\text{fullySynced}(\langle\rangle) &\overset{def}{\iff} \top \\
\text{fullySynced}(e^\frown es) &\overset{def}{\iff} \neg\,\text{unsynced}(e) \wedge \text{fullySynced}(es)
\end{aligned}$$

Figure 3.11: Predicates defining termination of the execution.

execution of a process when the PC is not in its domain. Every other inference rule is only applicable when the PC of the starting state is defined by the process.

The inference rules for singleton processes (BSDO, BSBR, BSCBR_T, BSCBR_F, BSRECV, BSSEND) define the behaviour of each instruction, and are analogous to the small-step transition rules.

The behaviour of composite processes is defined by the rules BSSEQL and BSSEQR. These rules are based on the observation that any execution of the composite process will begin on one of the two subprocesses, proceeding within the subprocess until the PC is no longer defined in it. The PC may now be in the domain of the other subprocess, and execution will continue from there and may jump back into the first subprocess. The execution of a loop between two subprocess can, therefore, be inferred by alternating BSSEQL and BSSEQR until the PC is no longer defined in either subprocess.

An important property of the big-step transition relation is that the only registers modified by it are those in the set of registers of the process. This is expressed by the following lemma, which was formalized in Isabelle.

Lemma 3.3. $s \underset{proc}{\overset{t}{\rightarrowtail}} s' \Longrightarrow \text{framedChange}_\text{s}(s, s', \text{regs}_\text{s}(proc))$

Proof. By induction on the derivation of the big-step transition.

The case for BSNOP is trivial, since no modification occurs.

The cases of singleton programs are analogous to the proof of framing for the small-step relation.

The cases for BSSEQL and BSSEQR are very similar, we proceed by only proving the former.

From the induction hypothesis, we have an intermediate register store σ'', such that

$$\frac{l \notin \mathrm{dom}(proc)}{(\sigma, l) \underset{proc}{\overset{\langle\rangle}{\rightarrowtail}} (\sigma, l)}\ \textsc{BSnop} \qquad \frac{\sigma' \in f(\sigma)}{(\sigma, l) \underset{(l,\mathtt{do}\, f)}{\overset{\langle\tau\rangle}{\rightarrowtail}} (\sigma', \mathrm{succ}\, l)}\ \textsc{BSdo} \qquad \frac{l \neq l'}{(\sigma, l) \underset{(l,\mathtt{br}\, l')}{\overset{\langle\tau\rangle}{\rightarrowtail}} (\sigma, l')}\ \textsc{BSbr}$$

$$\frac{b(\sigma) \qquad l \neq l'}{(\sigma, l) \underset{(l,\mathtt{cbr}\, b\, l')}{\overset{\langle\tau\rangle}{\rightarrowtail}} (\sigma, l')}\ \textsc{BScbr\_t} \qquad \frac{\neg b(\sigma)}{(\sigma, l) \underset{(l,\mathtt{cbr}\, b\, l')}{\overset{\langle\tau\rangle}{\rightarrowtail}} (\sigma, \mathrm{succ}\, l)}\ \textsc{BScbr\_f}$$

$$\frac{}{(\sigma, l) \underset{(l,\mathtt{recv}\, f\, c)}{\overset{\langle c?x\rangle}{\rightarrowtail}} (f(x, \sigma), \mathrm{succ}\, l)}\ \textsc{BSrecv} \qquad \frac{}{(\sigma, l) \underset{(l,\mathtt{send}\, f\, c)}{\overset{\langle c!f(\sigma)\rangle}{\rightarrowtail}} (\sigma, \mathrm{succ}\, l)}\ \textsc{BSsend}$$

$$\frac{l \in \mathrm{dom}(proc_1) \qquad (\sigma, l) \underset{proc_1}{\overset{t_1}{\rightarrowtail}} (\sigma'', l'') \qquad (\sigma'', l'') \underset{proc_1 \oplus proc_2}{\overset{t_2}{\rightarrowtail}} (\sigma', l')}{(\sigma, l) \underset{proc_1 \oplus proc_2}{\overset{t_1 \frown t_2}{\rightarrowtail}} (\sigma', l')}\ \textsc{BSseql}$$

$$\frac{l \in \mathrm{dom}(proc_2) \qquad (\sigma, l) \underset{proc_2}{\overset{t_1}{\rightarrowtail}} (\sigma'', l'') \qquad (\sigma'', l'') \underset{proc_1 \oplus proc_2}{\overset{t_2}{\rightarrowtail}} (\sigma', l')}{(\sigma, l) \underset{proc_1 \oplus proc_2}{\overset{t_1 \frown t_2}{\rightarrowtail}} (\sigma', l')}\ \textsc{BSseqr}$$

Figure 3.12: Big-Step relation for sequential processes.

both the changes from σ to σ'' and those from σ'' to σ' are framed as required. Clearly, the changes between σ and σ' must also be framed — if they weren't, then they either would already be present in σ'' and the first changes wouldn't be framed, or the second changes wouldn't be framed.

□

The big-step semantics of programs is presented in Figure 3.13. The rule for singleton programs (BSone) is similar to the small-step semantics, and merely promotes the execution of the process into an execution of the program. The rule for a composite program (BSpar) combines the executions of both subprograms, as long as the traces generated by them can be synchronized with each other. Since the only means of communication between processes (and subprograms) is through events, the traces provide all necessary information to ensure the two executions interact correctly, and the changes to PCs and registers may be simply combined.

Analogously to the transitions of sequential processes, small-step transitions of

$$\frac{(\sigma, \pi(p)) \underset{prog}{\overset{t}{\rightarrowtail}} (\sigma', l)}{(\sigma, \pi) \underset{p ::_c prog}{\overset{t}{\rightarrowtail}} (\sigma', \pi[p := l])}\ \textsc{BSone}$$

$$\frac{\begin{array}{c}(\sigma, \pi) \underset{prog_1}{\overset{t_1}{\rightarrowtail}} (\sigma_1, \pi_1) \qquad (\sigma, \pi) \underset{prog_2}{\overset{t_2}{\rightarrowtail}} (\sigma_2, \pi_2) \\ \mathrm{pids}(prog_1) \cap \mathrm{pids}(prog_2) = \emptyset \\ \mathrm{regs_c}(prog_1) \cap \mathrm{regs_c}(prog_2) = \emptyset \qquad t \in t_1 \parallel t_2 \\ \sigma' = \mathrm{update}(\sigma, \mathrm{regs_c}(prog_1), \sigma_1, \mathrm{regs_c}(prog_2), \sigma_2) \\ \pi' = \mathrm{update}(\pi, \mathrm{pids}(prog_1), \pi_1, \mathrm{pids}(prog_2), \pi_2)\end{array}}{(\sigma, \pi) \underset{prog_1 \parallel prog_2}{\overset{t}{\rightarrowtail}} (\sigma', \pi')}\ \textsc{BSpar}$$

Figure 3.13: Big-step relation for concurrent programs.

concurrent programs are also framed. The following lemma was also proved in Isabelle.

Lemma 3.4. $s \underset{prog}{\overset{t}{\rightarrowtail}} s' \Longrightarrow \mathrm{framedChange_c}(s, s', \mathrm{pids}(prog), \mathrm{regs_c}(prog))$

Proof. We prove by induction on the derivation of the transition. The case for BSone is a trivial consequence of the fact that big-step transitions for sequential processes are framed.

In the case of BSpar, we need to show that changes to both the process table and to the register store are framed. We have the following assumptions:

$$\begin{array}{c}prog = prog_1 \parallel prog_2 \\ (\sigma, \pi) \underset{prog_1}{\overset{t_1}{\rightarrowtail}} (\sigma_1, \pi_1) \ \wedge \ (\sigma, \pi) \underset{prog_2}{\overset{t_2}{\rightarrowtail}} (\sigma_2, \pi_2) \\ \sigma' = \mathrm{update}(\sigma, \mathrm{regs_s}(prog_1), \sigma_1, \mathrm{regs_s}(prog_2), \sigma_2) \\ \pi' = \mathrm{update}(\sigma, \mathrm{pids}(prog_1), \pi_1, \mathrm{pids}(prog_2), \pi_2)\end{array}$$

The changes caused by the transition are all caused by either of the sub-transitions, which are (by the induction hypothesis) framed to the registers of their respective subprograms. Since these are all contained in the registers of the complete program, all the changes must also be framed as required.

The argument for the framing of process tables is analogous.

□

In order to combine the traces of both executions we define the set of traces that may result from the synchronization of two others, denoted by $- \parallel -$ in Figure 3.14. It is defined as an inductive predicate and essentially allows arbitrary interleavings and synchronizations between the original events, ensuring the synchronization is correct via the partial function $- \parallel - : \text{event} \times \text{event} \longrightarrow \text{event}$.

The synchronization function for events is only defined when one of the events sends and the other receives, and they agree on both channel and value. The result is then the synchronization event with the same channel and value.

The synchronization predicate for traces allows the occurrence of empty traces (SYNCSempty), and allows independent steps on either side (SYNCSleft and SYNCSright). It also allows synchronization of single events, as long as they occur at the same time and (SYNCSsync). Note that interleaving may occur even when synchronization was an option (e.g. $\langle c!v, c?v\rangle \in \langle c!v\rangle \parallel \langle c?v\rangle$). This is necessary since multiple processes may communicate through the same channel and therefore, when analysing a subprogram, we cannot know if the event should synchronize within this subprogram or with another part of the program. The big-step transition of a program may therefore not represent a terminating execution, since the resulting traces may contain unfulfilled communication.

$$\frac{}{c.v = c!v \parallel c?v} \qquad \frac{}{c.v = c?v \parallel c!v}$$

$$\frac{}{\langle\rangle \in (\langle\rangle \parallel \langle\rangle)}\ \text{SYNCSempty}$$

$$\frac{zs \in xs \parallel ys}{\langle x\rangle^\frown zs \in (\langle x\rangle^\frown xs \parallel ys)}\ \text{SYNCSleft} \qquad \frac{zs \in xs \parallel ys}{\langle y\rangle^\frown zs \in (xs \parallel \langle y\rangle^\frown ys)}\ \text{SYNCSright}$$

$$\frac{z = x \parallel y \qquad zs \in xs \parallel ys}{\langle z\rangle^\frown zs \in (\langle x\rangle^\frown xs \parallel \langle y\rangle^\frown ys)}\ \text{SYNCSsync}$$

Figure 3.14: Synchronization predicates for traces and events.

3.4.3 Equivalence of the Presentations

It is important the the two presentations of LLDo's semantics are, in some sense, equivalent. The small-step semantics is more general, since it allows reasoning

about executions that did not yet terminate, while the big-step semantics only specifies the stuck executions. In this section, we prove Preservation and Reflection lemmas, stating that big-step transitions are equivalent to stuck multi-step transitions. All programs in this section are assumed well-formed, and all proofs were formalized in Isabelle.

Most of the important steps for proving the equivalence are properties of multi-step transitions, which we now provide in the following lemmas.

Lemma 3.5 (Singleton Transition)**.**

$$s \xrightarrow[p]{e} s' \Longrightarrow s \xrightarrow[p]{\langle e \rangle}{}^{*} s'$$

A small-step transition may be promoted to a multi-step transition. This holds for both sequential processes and concurrent programs.

Lemma 3.6 (Concatenation of Transitions)**.**

$$s \xrightarrow[p]{t}{}^{*} s' \wedge s' \xrightarrow[p]{t'}{}^{*} s'' \Longrightarrow s \xrightarrow[p]{t^\frown t'}{}^{*} s''$$

Two multi-step transitions may be "concatenated" when the final state of the first transition and the initial state of the second coincide. This holds for both sequential processes and concurrent programs.

Lemma 3.7 (Transitions of Subprocesses)**.**

$$s \xrightarrow[proc_1]{t}{}^{*} s' \vee s \xrightarrow[proc_2]{t}{}^{*} s' \Longrightarrow s \xrightarrow[proc_1 \oplus proc_2]{t}{}^{*} s'$$

If a multi-step transition may occur in a process, adding more instructions to it does not invalidate the transition.

Lemma 3.8 (Transitions of Subprograms)**.**

$$s \xrightarrow[prog_1]{t}{}^{*} s' \vee s \xrightarrow[prog_2]{t}{}^{*} s' \Longrightarrow s \xrightarrow[prog_1 \| prog_2]{t}{}^{*} s'$$

If a multi-step transition may occur in a program, adding more processes to it does not invalidate the transition.

Lemma 3.9 (Transitions of Singleton Programs)**.**

$$(\sigma, \pi) \xrightarrow[(p,proc)]{t}{}^{*} (\sigma', \pi') \Longrightarrow (\sigma, \pi(p)) \xrightarrow[proc]{t}{}^{*} (\sigma', \pi'(p))$$

A multi-step transition of a singleton program can only be caused by a multi-step transition of its single process.

Lemma 3.10 (Transition from an Invisibly Different State)**.**

$$R \cap \mathrm{regs_c}(\mathit{prog}) = \emptyset \ \wedge \ P \cap \mathrm{pids}(\mathit{prog}) = \emptyset \ \wedge$$
$$(\sigma, \pi) \xrightarrow[\mathit{prog}]{t}{}^{*} (\sigma', \pi')$$
$$\Longrightarrow (\mathrm{update}(\sigma, R, \sigma''), \mathrm{update}(\pi, P, \pi'')) \xrightarrow[\mathit{prog}]{t}{}^{*} (\mathrm{update}(\sigma', R, \sigma''), \mathrm{update}(\pi', P, \pi''))$$

Changes to the initial state of a transition that are framed outside *of the registers used by the program do not affect the transition.*

Lemma 3.11.

$$(\sigma, \pi) \xrightarrow[\mathit{prog}_1 \| \mathit{prog}_2]{t}{}^{*} (\sigma', \pi') \Longrightarrow \exists\, \sigma_1\, \sigma_2\, \pi_1\, \pi_2\, t_1\, t_2.$$
$$(\sigma, \pi) \xrightarrow[\mathit{prog}_1]{t_1}{}^{*} (\sigma_1, \pi_1) \ \wedge \ (\sigma, \pi) \xrightarrow[\mathit{prog}_2]{t_2}{}^{*} (\sigma_2, \pi_2) \ \wedge$$
$$\sigma' = \mathrm{update}(\sigma, \mathrm{regs_c}(\mathit{prog}_1), \sigma_1, \mathrm{regs_c}(\mathit{prog}_2), \sigma_2) \ \wedge$$
$$\pi' = \mathrm{update}(\pi, \mathrm{pids}(\mathit{prog}_1), \pi_1, \mathrm{pids}(\mathit{prog}_2), \pi_2) \ \wedge$$
$$t \in t_1 \parallel t_2$$

A multi-step transition of a parallel composition may be split into the transitions of each subprogram.

Proof. By induction on the derivation of the multi-step transition.

- If the last step was an application of MSZERO, we have $\sigma_i = \sigma$, $\pi_i = \pi$ and $t_i = \langle\rangle$, for $i \in \{1, 2\}$. Both multi-step transitions for prog_1 and prog_2 may be obtained by MSZERO, the synchronization of the traces by SYNCSEMPTY, the remaining goals are trivial.

- If the last step was an application of MSSTEP, we have $t = \langle e \rangle^\frown t'$ and the following transitions.

$$(\sigma, \pi) \xrightarrow[\mathit{prog}_1 \| \mathit{prog}_2]{e} (\sigma'', \pi'') \qquad\qquad (\sigma'', \pi'') \xrightarrow[\mathit{prog}_1 \| \mathit{prog}_2]{t'}{}^{*} (\sigma'', \pi'')$$

  Furthermore, we have the following induction hypothesis.

$$(\sigma'', \pi'') \xrightarrow[\mathit{prog}_1]{t_1'}{}^{*} (\sigma_1, \pi_1) \ \wedge \ (\sigma'', \pi'') \xrightarrow[\mathit{prog}_2]{t_2'}{}^{*} (\sigma_2, \pi_2) \ \wedge \ t' \in t_1' \parallel t_2'$$
$$\sigma' = \mathrm{update}(\sigma, \mathrm{regs_c}(\mathit{prog}_1), \sigma_1, \mathrm{regs_c}(\mathit{prog}_2), \sigma_2)$$
$$\pi' = \mathrm{update}(\pi, \mathrm{pids}(\mathit{prog}_1), \pi_1, \mathrm{pids}(\mathit{prog}_2), \pi_2)$$

We proceed by cases on the derivation of the small-step transition of $prog_1 \parallel prog_2$.

– If the last step was an application of SSOne, the transition occurred in a single process, which is either part of $prog_1$ or of $prog_2$. Without loss of generality, we assume the former, obtaining the following.

$$(\sigma, \pi) \xrightarrow[prog_1]{e} (\sigma'', \pi'')$$

The complete transition for $prog_1$ is obtained by MSStep, and the synchronization of the traces by SYNCSLeft. The transition for $prog_2$ may be started from (σ'', π''), since any differences are framed to $prog_1$ (lemma 3.10). Also because of this framing, the merged state remains the same.

– If the last step was a application of SSSync, the transition occurred in two processes. If those processes are both part of $prog_1$, or both part of $prog_2$, the proof is analogous to the previous case. If each of the subprograms contains one of the executing processes, we obtain the following.

$$\begin{gathered}(\sigma, \pi) \xrightarrow[prog_1]{c!v} (\sigma_1', \pi_1') \ \wedge \ (\sigma, \pi) \xrightarrow[prog_2]{c?v} (\sigma_2', \pi_2') \ \wedge \ e = e_1 \parallel e_2 \\ \sigma'' = \text{update}(\sigma, \text{regs}_\text{c}(prog_1), \sigma_1', \text{regs}_\text{c}(prog_2), \sigma_2') \\ \pi'' = \text{update}(\pi, \text{pids}(prog_1), \pi_1', \text{pids}(prog_2), \pi_2')\end{gathered}$$

The complete transitions for each subprogram are obtained by MSStep. We note that the multi-step transitions may be started from (σ_i, π_i) instead of from (σ'', π''), for $i \in 1, 2$, since any differences are framed to the opposite subprocess (lemma 3.10). Also because of this framing, the final merged state remains the same. The synchronization of the traces may be obtained by SYNCSSync.

□

Having those lemmas, we may now prove the Preservation and Reflection lemmas.

Lemma 3.12 (Preservation of Big-Step Semantics for Processes)**.** *If the big-step transition* $s \overset{t}{\underset{proc}{\rightarrowtail}} s'$ *of the sequential process proc is valid, then the multi-step transition* $s \xrightarrow[proc]{t}{}^{*} s'$ *is also valid.*

Proof. By induction on the derivation of $s \overset{t}{\underset{proc}{\rightarrowtail}} s'$. We have the following cases.

- The last rule of the derivation is BSNOP. The small-step transition may be trivially obtained by MSZERO.

- The last rule of the derivation is one of BSDO, BSBR, BSCBR_T, BSCBR_F, BSSEND, BSRECV. The small-step transition may be trivially obtained by using an analogous inference rule and by lemma 3.5.

- The last rule of the derivation is BSSEQL or BSSEQR. Without loss of generality, we assume it was BSSEQL. By the induction hypothesis, the following transitions are valid.

  $$s \underset{proc_1}{\overset{t}{\longrightarrow}}{}^{*} s'' \qquad\qquad s'' \underset{proc_1 \oplus proc_2}{\overset{t'}{\longrightarrow}}{}^{*} s'$$

  The transition on $proc_1$ is also valid for the combined process (lemma 3.7).

  $$s \underset{proc_1 \oplus proc_2}{\overset{t}{\longrightarrow}}{}^{*} s''$$

  The two transitions may be combined to obtain our goal (lemma 3.6).

  $$s \underset{proc_1 \oplus proc_2}{\overset{t \frown t'}{\longrightarrow}}{}^{*} s'$$

□

Lemma 3.13 (Reflection of Big-Step Semantics for Processes)**.** *If the step-indexed multi-step transition* $s \overset{t,\,i}{\underset{proc}{\longrightarrow}}{}^{*} s'$ *of the sequential process proc is valid and the evaluation is stuck, i.e.* $\text{stuck}_s(s, proc)$*, then the big-step transition* $s \overset{t}{\underset{proc}{\rightarrowtail}} s'$ *is also valid.*

Proof. We begin by observing that if $i = 0$, the only possible multi-step transition is the empty one. In this case, $s' = s$ and $PC(s) \notin \text{dom}(proc)$, therefore the big-step transition is trivially obtained by BSNOP.

We proceed the proof with the case where $i > 0$, by structural induction on *proc*.

- When $proc = \emptyset$, the only possible multi-step transition is the empty transition, $i > 0$ cannot occur.

- When we have a singleton process $proc = (l, instr)$, we consider two cases for the number of steps i taken on the multi-step transition.

  - When $i = 1$, the multi-step transition corresponds to a single small-step transition, which is derived by one of SSDO, SSBR, SSCBR_T, SSCBR_F, SSSEND, SSRECV. The big-step transition can be obtained by the analogous big-step inference rule.

  - We cannot have $i > 1$. In all inference rules of the small-step relation but SSBR and SSCBR_T, the PC of the second state is $l + 1$, which is not defined by $(l, instr)$ — there can be no second step. In the two other rules, the PC of the second state is some l'. If $l' \neq l$, the PC of the second state is not defined by the process and there can be no second step. If $l' = l$, the PC of the following states will *always* be l, and in particular $PC(s')$. This would contradict the hypothesis that the multi-step transition was stuck.

- When $proc = proc_1 \oplus proc_2$, we obtain the following induction hypothesis.

$$\forall\, s\ t\ s'\ k. \qquad s \xrightarrow[proc_1]{t,k}{}^{*}\ s' \wedge \mathrm{stuck_s}(s', proc_1) \Longrightarrow s \overset{t}{\underset{proc_1}{\rightarrowtail}}\ s'$$

$$\forall\, s\ t\ s'\ k. \qquad s \xrightarrow[proc_2]{t,k}{}^{*}\ s' \wedge \mathrm{stuck_s}(s', proc_2) \Longrightarrow s \overset{t}{\underset{proc_2}{\rightarrowtail}}\ s'$$

The proof follows by strong induction on the number i of steps taken in the multi-step transition. We further obtain the following induction hypothesis.

$$\forall\, s\ t\ s'\ m. \qquad m < i \ \wedge\ s \xrightarrow[proc_1 \oplus proc_2]{t,m}{}^{*}\ s' \ \wedge\ \mathrm{stuck_s}(s', proc_1 \oplus proc_2)$$
$$\Longrightarrow\ s \overset{t}{\underset{proc_1 \oplus proc_2}{\rightarrowtail}}\ s'$$

Since $i > 0$, there must exist a small-step transition on $proc$ starting with state $s = (\sigma, l)$, therefore $l \in \mathrm{dom}(proc)$. Since each label may only be defined in one of the subprocesses, we have two cases, $l \in \mathrm{dom}(proc_1)$ or $l \in \mathrm{dom}(proc_2)$, and assume the former without loss of generality.

At least one step must have been executed in $proc_1$, followed by another multi-step transition on the combined process. Therefore, there must exist

j_1, j_2, s'', t_1, t_2 such that $j_1 > 0$, $j_1 + j_2 = i$ and the following multi-step transitions are valid.

$$s \xrightarrow[proc_1]{t_1, j_1}{}^{*} s'' \qquad\qquad s'' \xrightarrow[proc_1 \| proc_2]{t_2, j_2}{}^{*} s'$$

A big-step transition on $proc_1$ may then be obtained by applying one of the outer induction hypothesis, whereas the transition on the combined program may be obtained by applying the induction hypothesis from the strong natural induction, noting that $j_2 < j_1 + j_2$. The resulting big-step transitions may be used with BSAPPENDL to obtain the goal.

□

Lemma 3.14 (Preservation of Big-Step Semantics for Programs)**.** *If the big-step transition* $(\sigma, \pi) \overset{t}{\underset{prog}{\rightarrowtail}} (\sigma', \pi')$ *of the concurrent program prog is valid, then the multi-step transition* $(\sigma, \pi) \xrightarrow[prog]{t}{}^{*} (\sigma', \pi')$ *is also valid.*

Proof. By induction on the derivation of the big-step transition.

- If the last step of the derivation was an application of BSONE, we have the following assumptions:

  $$prog = (p, proc) \qquad (\sigma, \pi(p)) \overset{t}{\underset{proc}{\rightarrowtail}} (\sigma', l') \qquad \pi' = \pi[p := l']$$

  By preservation for processes, an equivalent multi-step transition can be inferred.

  $$(\sigma, \pi(p)) \xrightarrow[proc]{t}{}^{*} (\sigma', l')$$

  We now need to "promote" this transition of the process to a transition of the complete program, proceeding by induction on the derivation of the multi-step transition.

  – If the last step of the derivation was an application of MSZERO, we have $\sigma' = \sigma$ and $l' = \pi(p)$, therefore $\pi' = \pi$. The equivalent transition of the complete program can be derived by MSZERO.

  – If the last step of the derivation was an application of MSSTEP, we have $t = \langle e \rangle^\frown t'$ and the following transition, which may be made into a transition of the complete program via SSONE.

$$(\sigma, \pi(p)) \xrightarrow[proc]{e} (\sigma'', l'')$$

Furthermore, we have the following induction hypothesis.

$$(\sigma'', \pi[p := l'']) \xrightarrow[(p,proc)]{t'}{}^{*} (\sigma'', l'')$$

This and the small-step transition may be combined via MSSTEP into a transition of the complete program.

- If the last step of the derivation was an application of BSPAR, we have the following induction hypothesis.

$$(\sigma, \pi) \xrightarrow[prog_1]{t_1}{}^{*} (\sigma_1, \pi_1) \qquad (\sigma, \pi) \xrightarrow[prog_2]{t_2}{}^{*} (\sigma_2, \pi_2)$$

Furthermore, the following facts were necessary for the application of BSPAR.

$$prog = prog_1 \parallel prog_2 \qquad t \in t_1 \parallel t_2$$

$$\sigma' = \text{update}(\sigma, \text{regs}_\text{c}(prog_1), \sigma_1, \text{regs}_\text{c}(prog_2), \sigma_2)$$

$$\pi' = \text{update}(\pi, \text{pids}(prog_1), \pi_1, \text{pids}(prog_2), \pi_2)$$

We proceed by induction on the derivation of $t \in t_1 \parallel t_2$.

  - If the last step of the derivation was an application of SYNCSEMPTY, we have $t_1 = t_2 = t = \langle\rangle$. The only possible multi-step transition with empty trace is the empty transition, therefore we have $\sigma' = \sigma_1 = \sigma_2 = \sigma$ and $\pi' = \pi_1 = \pi_2 = \pi$. The multi-step transition of the combined program is obtained by applying MSZERO.

  - If the last step of the derivation was an application of SYNCSLEFT or SYNCSRIGHT, we assume the former without loss of generality. We have $t_1 = \langle e \rangle^\frown t_1'$, $t = \langle e \rangle^\frown t'$, $t' \in t_1' \parallel t_2$ and the following induction hypothesis. It states that, if there exist two multi-step transitions on $prog_1$ and $prog_2$ from the same original state, that yield the traces t_1' and t_2 respectively, then there exists a combined transition yielding the trace t'.

$$\begin{aligned}
&\forall\, \sigma\, \sigma'\, \sigma_1\, \sigma_2\, \pi\, \pi'\, \pi_1\, \pi_2.\\
&\qquad (\sigma, \pi) \xrightarrow[prog_1]{t_1'}{}^{*} (\sigma_1, \pi_1) \;\wedge\; (\sigma, \pi) \xrightarrow[prog_2]{t_2}{}^{*} (\sigma_2, \pi_2) \;\wedge\\
&\qquad \sigma' = \text{update}(\sigma, \text{regs}_\text{c}(prog_1), \sigma_1, \text{regs}_\text{c}(prog_2), \sigma_2) \;\wedge\\
&\qquad \pi' = \text{update}(\pi, \text{pids}(prog_1), \pi_1, \text{pids}(prog_2), \pi_2)\\
&\Longrightarrow (\sigma, \pi) \xrightarrow[prog_1 \parallel prog_2]{t'}{}^{*} (\sigma'', \pi'')
\end{aligned}$$

In order to apply the induction hypothesis, we need to find the individual multi-step transitions. We note that the only way to obtain the trace $\langle e \rangle^\frown t_1'$ is if the multi-step transition on $prog_1$ was obtained from the following transitions.

$$(\sigma, \pi) \xrightarrow[prog_1]{e} (\sigma'', \pi'') \qquad (\sigma'', \pi'') \xrightarrow[prog_1]{t_1'}{}^* (\sigma_1, \pi_1)$$

Noting that the transition for $prog_2$ may also be started on (σ'', π'') (lemma 3.10), since any differences are framed by $prog_1$, we apply the induction hypothesis to obtain the following.

$$(\sigma'', \pi'') \xrightarrow[prog_1 \parallel prog_2]{t'}{}^* (\sigma', \pi')$$

Since the small-step transition on $prog_1$ is also valid for the whole program (lemma 3.8), we may use MSStep to combine it and to the multi-step transition above, proving our goal.

– If the last step of the derivation was an application of SYNCSSync, we have the following assumptions.
$t_1 = \langle e_1 \rangle^\frown t_1' \qquad t_2 = \langle e_2 \rangle^\frown t_2' \qquad t = \langle e \rangle^\frown t' \qquad t' \in t_1' \parallel t_2' \qquad e = e_1 \parallel e_2$
Furthermore, we have the following induction hypothesis. It states that, if there exist two multi-step transitions on $prog_1$ and $prog_2$ from the same original state, that yield the traces t_1' and t_2' respectively, then there exists a combined transition yielding the trace t'.

$$\begin{aligned}
&\forall\, \sigma\, \sigma'\, \sigma_1\, \sigma_2\, \pi\, \pi'\, \pi_1\, \pi_2. \\
&\qquad (\sigma, \pi) \xrightarrow[prog_1]{t_1'}{}^* (\sigma_1, \pi_1) \;\wedge\; (\sigma, \pi) \xrightarrow[prog_2]{t_2'}{}^* (\sigma_2, \pi_2) \;\wedge \\
&\qquad \sigma' = \mathrm{update}(\sigma, \mathrm{regs_c}(prog_1), \sigma_1, \mathrm{regs_c}(prog_2), \sigma_2) \;\wedge \\
&\qquad \pi' = \mathrm{update}(\pi, \mathrm{pids}(prog_1), \pi_1, \mathrm{pids}(prog_2), \pi_2) \\
&\Longrightarrow (\sigma, \pi) \xrightarrow[prog_1 \parallel prog_2]{t'}{}^* (\sigma'', \pi'')
\end{aligned}$$

In order to apply the induction hypothesis, we need to find the individual multi-step transitions. We note that the only way to obtain the traces $\langle e_1 \rangle^\frown t_1'$ and $\langle e_2 \rangle^\frown t_2'$ is if the multi-step transitions of $prog_1$ and $prog_2$ were obtained from the following pairs of transitions.

$$(\sigma, \pi) \xrightarrow[prog_1]{e_1} (\sigma_1', \pi_1') \qquad (\sigma_1', \pi_1') \xrightarrow[prog_1]{t_1'}{}^* (\sigma_1, \pi_1)$$

$$(\sigma, \pi) \xrightarrow[prog_2]{e_2} (\sigma_2', \pi_2') \qquad (\sigma_2', \pi_2') \xrightarrow[prog_2]{t_2'}{}^* (\sigma_2, \pi_2)$$

We will now obtain a joint small-step transition into the state (σ'', π''), constructed as follows.

$$\sigma'' = \text{update}(\sigma, \text{regs}_c(prog_1), \sigma_1', \text{regs}_c(prog_2), \sigma_2')$$
$$\pi'' = \text{update}(\pi, \text{pids}(prog_1), \pi_1', \text{pids}(prog_2), \pi_2')$$

We note that, since $e = e_1 \parallel e_2$, one of e_1 and e_2 must be a sending event and the other, a receiving one. Therefore, the small-step transitions that trigger them cannot have been inferred with SSSYNC, only with SSONE. The following must hold.

$$(p_1, proc_1) \in prog_1 \ \wedge \ (\sigma_1', \pi_1'(p_1)) \xrightarrow[proc_1]{e_1} (\sigma_1, l_1) \ \wedge \ \pi_1 = \pi_1'[p_1 := l_1]$$
$$(p_2, proc_2) \in prog_2 \ \wedge \ (\sigma_2', \pi_2'(p_2)) \xrightarrow[proc_2]{e_2} (\sigma_2, l_2) \ \wedge \ \pi_2 = \pi_2'[p_2 := l_2]$$

The combined small-step transition may be obtained from SSSYNC, since it is also clearly true that $(p_1, proc_1) \in prog_1 \parallel prog_2$ and that $(p_2, proc_2) \in prog_1 \parallel prog_2$.

$$(\sigma, \pi) \xrightarrow[prog_1 \parallel prog_2]{e} (\sigma'', \pi'')$$

Noting that the remaining multi-step transitions may also be started on (σ'', π''), since any differences are framed by the opposite program (lemma 3.10), we apply the induction hypothesis to obtain the following.

$$(\sigma'', \pi'') \xrightarrow[prog_1 \parallel prog_2]{t'}{}^{*} (\sigma', \pi')$$

We may use MSSTEP to combine the small- and multi-step transitions above, proving our goal.

□

Lemma 3.15 (Reflection of Big-Step Semantics for Programs)**.** *If the multi-step transition* $(\sigma, \pi) \xrightarrow[prog]{t}{}^{*} (\sigma', \pi')$ *of the concurrent program prog is valid and* $\text{stuck}_c(\sigma, prog)$*, then the big-step transition* $(\sigma, \pi) \overset{t}{\underset{prog}{\rightarrowtail}} (\sigma', \pi')$ *is also valid.*

Proof. By structural induction on *prog*. We have the following cases:

- When the program consists of a single process, $prog = (p, proc)$, the transition may only be introduced by BSONE. It remains to show that the following big-step transition of the process is valid.

$$(\sigma, \pi(p)) \overset{t}{\underset{proc}{\rightarrowtail}} (\sigma', \pi'(p))$$

The equivalent multi-step transition of the process may be obtained, from the multi-step transition of the program, by the lemma 3.9. We conclude this case by applying the reflection lemma for processes.

- When the program is composite, $prog = prog_1 \parallel prog_2$, we have the following induction hypothesis.

$$\forall\, \sigma\ \pi\ t\ \sigma'\ \pi'. \qquad (\sigma, \pi) \xrightarrow[prog_1]{t}{}^{*} (\sigma', \pi') \wedge \mathrm{stuck_c}(\sigma', prog_1) \Longrightarrow (\sigma, \pi) \overset{t}{\underset{prog_1}{\rightarrowtail}} (\sigma', \pi')$$

$$\forall\, \sigma\ \pi\ t\ \sigma'\ \pi'. \qquad (\sigma, \pi) \xrightarrow[prog_2]{t}{}^{*} (\sigma', \pi') \wedge \mathrm{stuck_c}(\sigma', prog_2) \Longrightarrow (\sigma, \pi) \overset{t}{\underset{prog_2}{\rightarrowtail}} (\sigma', \pi')$$

The big-step transition for $prog_1 \parallel prog_2$ will be introduced by BSpar. Two of its premises, the big-step transitions for $prog_1$ and for $prog_2$, may be obtained from the induction hypothesis. We need, therefore, to prove the validity of the following independent multi-step transitions, for some resulting state (σ_1, π_1) and some (σ_2, π_2).

$$(\sigma, \pi) \xrightarrow[prog_1]{t_1}{}^{*} (\sigma_1, \pi_1) \qquad\qquad (\sigma, \pi) \xrightarrow[prog_2]{t_2}{}^{*} (\sigma_2, \pi_2)$$

Furthermore, the other premises of BSpar must hold. The disjointness conditions are ensured by the well-formedness of the programs, but the following also remain to be proved.

$$t \in t_1 \parallel t_2$$
$$\sigma' = \mathrm{update}(\sigma, \mathrm{regs_c}(prog_1), \sigma_1, \mathrm{regs_c}(prog_2), \sigma_2)$$
$$\pi' = \mathrm{update}(\pi, \mathrm{pids}(prog_1), \pi_1, \mathrm{pids}(prog_2), \pi_2)$$

All these are obtained by applying the lemma 3.11

□

4 Correctness Calculus

Although the formal semantics provides a rigorous understanding of a programming language, it does not suffice for verifying the correctness of programs. One of the many approaches for doing it is by specifying the behaviour of the program with pre- and postconditions.

Such a specification is written as the Hoare triple $\{P\}code\{Q\}$, where P is an assertion that is assumed to hold before the execution of the *code*, and Q an assertion expected to hold after the terminating execution. Such assertions are simply predicates of the state and trace from a given moment.We write them as usual logical expressions, using the variable h for referring to the current trace, σ for referring to the current register store, pc to the current PC (when reasoning about sequential processes) and π to the current process table (when reasoning about concurrent programs). We also denote by $P[e/x]$ the replacement of the free occurrences of a variable x, within the assertion P, by the logical expression e.

Although using the operational semantics directly to prove the validity of a Hoare triple would be possible, it is not practical. Instead, we define a set of inference rules for proving that the program satisfies the specification without explicitly reasoning about all its possible executions. In this section, we provide such a set of inference rules for LLDo and prove its correctness regarding the big-step semantics.

4.1 Definition

The inference rules of the correctness calculus may be classified in three groups: those reasoning about sequential processes, those reasoning about concurrent programs and general rules that are independent of the code being analyzed, expressing rules of the logic itself. In the case of this thesis, there is only one general rule, which may be seen in Figure 4.1. The rule HCONSEQ postulates that preconditions may be strengthened and postconditions may be weakened without invalidating the correctness of a Hoare triple.

$$\frac{P \Longrightarrow P' \qquad \vdash \{P'\}\ code\ \{Q'\} \qquad Q' \Longrightarrow Q}{\vdash \{P\}\ code\ \{Q\}}\ \textsc{Hconseq}$$

Figure 4.1: The general rule for the correctness calculus.

The inference rules that deal with sequential processes may be seen in Figure 4.2. The rule for empty programs states that the precondition is preserved, since nothing can be executed. All rules for single instructions follow the same pattern, having their precondition determined by the postcondition: either the PC points to the current instruction and the postcondition will hold in any of the states that may be caused by the execution of the instruction, or the PC does not point to the current instruction and the postcondition holds in the original state, since nothing will be executed.

The rule for the union of processes must account for possible loops that may repeatedly alternate the execution between the two subprocesses. Therefore, it is based on an invariant: we require that both subprocesses of $proc_1 \oplus proc_2$, when started in a state that satisfies the invariant, result in a state that also satisfies it. Thus we may safely alternate between executing $proc_1$ and $proc_2$ and, when we are done, we know the invariant holds and we are outside the domains of both subprograms.

The inference rules that deal with concurrent programs may be seen in Figure 4.3. The rule HONE for singleton programs is a simple "lifting": the assertions remain the same, except they now refer explicitly to an entry of the process table.

The rule HPAR for the parallel composition is more complex. The pre- and postconditions of the conclusion are derived from those of the premises. The resulting precondition states that there are two traces whose synchronization results in the current trace h, and that the preconditions of both subprograms hold regarding their respective traces. The postconditon is constructed in the same manner. Furthermore, we require that the postconditions are framed to their respective subprograms — that is, they refer only to registers and processes used by the subprogram they specify.

4.2 Correctness

Since explicitly reasoning about all possible executions of a piece of code is not a practical way to verify that it satisfies a specification, we provided a proof calculus for inferring the validity of Hoare triples. In this section, we prove that inferred triples are indeed semantically valid. All proofs described in this section were formalized in the Isabelle theorem prover.

We begin by defining the semantic validity of a Hoare triple in Figure 4.4. It is essentially a formalization of the intuition behind pre- and postconditions: whenever the precondition holds in the initial state of execution, the postcondition holds after the execution. In terms of the traces, we ignore any τ events.

The following lemmas prove the correctness of the proof calculus for Hoare triples.

$$\frac{}{\vdash \{P\}\ \emptyset\ \{P\}}\ \textsc{Hempty} \qquad \frac{}{\vdash \left\{\begin{array}{l}(pc = l \wedge Q[l'/pc]) \\ \vee\ (pc \neq l \wedge Q)\end{array}\right\}\ l ::_s \texttt{br}\ l'\ \{Q\}}\ \textsc{Hbr}$$

$$\frac{}{\vdash \left\{\begin{array}{l}(pc = l \wedge \forall\, \sigma' \in f(\sigma).\, Q[\sigma'/\sigma, pc+1/pc]) \\ \vee\ (pc \neq l \wedge Q)\end{array}\right\}\ l ::_s \texttt{do}\, f\ \{Q\}}\ \textsc{Hdo}$$

$$\frac{}{\vdash \left\{\begin{array}{l}(pc = l \wedge P(\sigma) \wedge Q[l'/pc]) \\ \vee\ (pc = l \wedge \neg P(\sigma) \wedge Q[l'/pc]) \\ \vee\ (pc \neq l \wedge Q)\end{array}\right\}\ l ::_s \texttt{cbr}\, P\, l'\ \{Q\}}\ \textsc{Hcbr}$$

$$\frac{}{\vdash \left\{\begin{array}{l}(pc = l \wedge Q[pc+1/pc, h^\frown\langle c!f(\sigma)\rangle/h]) \\ \vee\ (pc \neq l \wedge Q)\end{array}\right\}\ l ::_s \texttt{send}\, f\ c\ \{Q\}}\ \textsc{Hsend}$$

$$\frac{}{\vdash \left\{\begin{array}{l}(pc = l \wedge \forall\, v.Q[f(v,\sigma)/\sigma, pc+1/pc, h^\frown\langle c?v\rangle/h]) \\ \vee\ (pc \neq l \wedge Q)\end{array}\right\}\ l ::_s \texttt{recv}\, f\ c\ \{Q\}}\ \textsc{Hrecv}$$

$$\frac{\begin{array}{l}\vdash \{pc \in dom(proc_1) \wedge P\}\ proc_1\ \{pc \notin dom(proc_1) \wedge P\} \\ \vdash \{pc \in dom(proc_2) \wedge P\}\ proc_2\ \{pc \notin dom(proc_2) \wedge P\}\end{array}}{\vdash \{P\}\ proc_1 \oplus proc_2\ \{pc \notin dom(proc_1 \oplus proc_2) \wedge P\}}\ \textsc{Happend}$$

Figure 4.2: The rules of the correctness calculus for reasoning about sequential processes.

Lemma 4.1 (Correctness of Calculus for Processes)**.**

$$\vdash \{P\}\ proc\ \{Q\} \Longrightarrow \vDash \{P\}\ proc\ \{Q\}$$

Proof. By induction on the derivation of the Hoare triple.

- If the last step was an application of Hempty, we have the same assertion as precondition and postcondition. Since the program is empty, the initial and final state of execution must be the same, therefore the triple is semantically valid.

$$\frac{\vdash \{P\}\ proc\ \{Q\}}{\vdash \{P[\pi(p)/pc]\}\ p ::_c proc\ \{Q[\pi(p)/pc]\}}\ \textsc{Hone}$$

$$\frac{\begin{array}{c}\vdash \{P_1\}\ prog_1\ \{Q_1\} \qquad \vdash \{P_2\}\ prog_2\ \{Q_2\} \\ \text{framedAssn}(Q_1, \text{regs}_c(prog_1), \text{pids}(prog_1)) \\ \text{framedAssn}(Q_2, \text{regs}_c(prog_2), \text{pids}(prog_2))\end{array}}{\vdash \left\{\begin{array}{r}\exists\, t_1\ t_2.\ h \in (t_1 \parallel t_2) \wedge \\ P_1[t_1/h] \wedge P_2[t_2/h]\end{array}\right\}\ prog_1 \parallel prog_2\ \left\{\begin{array}{r}\exists\, t_1\ t_2.\ h \in (t_1 \parallel t_2) \wedge \\ Q_1[t_1/h] \wedge Q_2[t_2/h]\end{array}\right\}}\ \textsc{Hpar}$$

$$\begin{aligned}\text{framedAssn}(Q, R, P) \stackrel{def}{=} \forall\, R'\ T'\ \sigma'\ \pi'.\ & R \cap R' = \emptyset \wedge P \cap P' = \emptyset \wedge \\ & \text{framedChange}(\sigma, \sigma', R) \wedge \text{framedChange}(\pi, \pi', P) \wedge \\ & Q \Longrightarrow Q[\sigma'/\sigma, \pi'/\pi]\end{aligned}$$

Figure 4.3: The rules of the correctness calculus for reasoning about concurrent programs.

- If the last step was an application of Hdo, we have the following assumptions.

$$(pc = l \wedge \forall\, \sigma' \in f(\sigma).Q[\sigma'/\sigma, pc + 1/pc]) \vee (pc \neq l \wedge Q)$$
$$(\sigma, pc) \underset{l ::_s \mathtt{do}\, f}{\overset{t}{\rightarrowtail}} (\sigma', l')$$

  It remains to show that the postcondition holds in the final state, that is, $Q[\sigma'/\sigma, l'/pc]$.

  If $pc \neq l$, the only big-step transition possible is the empty one, therefore we have $\sigma' = \sigma$, $l' = pc$ and $t = \langle\rangle$, therefore the our goal is $Q[\sigma'/\sigma, l'/pc, h/h] = Q$, which is also provided by the precondition when $pc \neq l$.

  If $pc = l$, the only big-step transition possible is derived from BSdo. Therefore, we have $\sigma' \in f(\sigma)$, $l' = pc + 1$ and $t = \langle\tau\rangle$. Again, our goal is to prove $Q[\sigma'/\sigma, pc + 1/pc, h/h] = Q[\sigma'/\sigma, pc + 1/pc]$, which is implied by the precondition when $pc = l$.

- If the last step was an application of Hbr, Hcbr, Hsend or Hrecv, the proof is analogous to the case of Hdo.

$$\models \{P\}\ proc\ \{Q\} \stackrel{def}{=}$$
$$\forall\, \sigma\, \sigma'\, pc\, l'\, h\, t.\ P \wedge (\sigma, pc) \underset{proc}{\overset{t}{\rightarrowtail}} (\sigma', l') \Longrightarrow$$
$$Q[\sigma'/\sigma, l'/pc, h^\frown \text{visibleTrace}(t)/h]$$

$$\models \{P\}\ prog\ \{Q\} \stackrel{def}{=}$$
$$\forall\, \sigma\, \sigma'\, \pi\, \pi'\, h\, t.\ P \wedge (\sigma, \pi) \underset{prog}{\overset{t}{\rightarrowtail}} (\sigma', \pi') \Longrightarrow$$
$$Q[\sigma'/\sigma, \pi'/\pi, h^\frown \text{visibleTrace}(t)/h]$$

$$\text{visibleTrace}(\langle\rangle) = \langle\rangle$$
$$\text{visibleTrace}(\langle e\rangle^\frown t) = \begin{cases} t & \text{if } e = \tau \\ \langle e\rangle^\frown t & \text{if } e \neq \tau \end{cases}$$

Figure 4.4: Definition of the semantic validity of a Hoare triple.

- If the last step was an application of HAPPEND, we have the following induction hypothesis.

$$\models \{pc \in \text{dom}(proc_1) \wedge P\}\ proc_1\ \{pc \notin \text{dom}(proc_1) \wedge P\}$$
$$\models \{pc \in \text{dom}(proc_2) \wedge P\}\ proc_2\ \{pc \notin \text{dom}(proc_2) \wedge P\}$$

  Furthermore, we may assume that P holds and that the following transition is valid.

$$(\sigma, pc) \underset{proc_1 \oplus proc_2}{\overset{t}{\rightarrowtail}} (\sigma', l')$$

  Since the final state of a big-step transition always has the PC outside the domain of the process, only the following fact remains to be shown.

$$P[\sigma'/\sigma, l'/l, h^\frown \text{visibleTrace}(t)/h]$$

  We proceed by induction on the derivation of the transition.

  – If the last derivation step was an application of BSAPPENDL or BSAPPENDR, we may assume without loss of generality that it was BSAPPENDL. We

assume the premises of the rule.

$$(\sigma, pc) \underset{prog_1}{\overset{t_1}{\rightarrowtail}} (\sigma'', l'')$$

$$(\sigma'', l'') \underset{prog_1 \oplus prog_2}{\overset{t_2}{\rightarrowtail}} (\sigma', l')$$

We get the following induction hypothesis:

$$\forall\, t. P[\sigma''/\sigma, l''/pc, t/h] \Longrightarrow P[\sigma'/\sigma, l'/pc, t^\frown \text{visibleTrace}(t_2)/h]$$

From the outer induction hypothesis and the big-step transition on $prog_1$, we get $P[\sigma''/\sigma, l''/pc, h^\frown \text{visibleTrace}(t_1)/h]$. Now the inner induction hypothesis suffices to prove our goal.

– If the last derivation step was an application of BSnop, the proof is trivial: since the state doesn't change, the invariant is maintained.

– No other rule could have been applied.

- If the last derivation step was an application of Hconseq, we get the following induction hypothesis.

$$\vDash \{P'\}\ proc\ \{Q'\}$$

Furthermore, we assume the following premises of the rule.

$$(P \Longrightarrow P') \wedge (Q' \Longrightarrow Q)$$

The proof is now trivial: if P holds in the first state of the execution, then P' holds as well. Therefore, Q' must hold in the final state of the execution, which implies that Q holds in the final state.

□

Lemma 4.2 (Visible Trace preserves Synchronization)**.**

$$t \in (t_1 \parallel t_2) \Longrightarrow \text{visibleTrace}(t) \in (\text{visibleTrace}(t_1) \parallel \text{visibleTrace}(t_2))$$

Lemma 4.3 (Concatenation of Synchronized Traces)**.**

$$t \in (t_1 \parallel t_2) \wedge t' \in (t_1' \parallel t_2') \Longrightarrow t^\frown t' \in (t_1^\frown t_1' \parallel t_2^\frown t_2')$$

Lemma 4.4 (Correctness of Calculus for Programs)**.**

$$\vdash \{P\}\ prog\ \{Q\} \Longrightarrow \vDash \{P\}\ prog\ \{Q\}$$

Proof. By induction on the derivation of the Hoare triple.

- If the last step of the derivation was an application of HONE, the proof is a simple consequence of the correctness for processes.

- If the last step of the derivation was an application of HPAR, we obtain the following induction hypothesis.

$$\vDash \{P_1\}\ prog_1\ \{Q_1\}$$
$$\vDash \{P_2\}\ prog_2\ \{Q_2\}$$

We assume the following precondition and big-step transition, for some t_1, t_2.

$$h \in (t_1 \parallel t_2) \wedge P_1[t_1/h] \wedge P_2[t_2/h]$$
$$(\sigma, \pi) \underset{prog_1 \parallel prog_2}{\overset{t}{\rightarrowtail}} (\sigma', \pi')$$

The only way to infer the big-step transition was with BSPAR, so its premises must hold.

$$(\sigma, \pi) \underset{prog_1}{\overset{t_1'}{\rightarrowtail}} (\sigma_1, \pi_1) \ \wedge\ (\sigma, \pi) \underset{prog_2}{\overset{t_2'}{\rightarrowtail}} (\sigma_2, \pi_2) \ \wedge\ t \in (t_1' \parallel t_2')$$
$$\sigma' = \text{update}(\sigma, \text{regs}_\text{c}(prog_1), \sigma_1, \text{regs}_\text{c}(prog_2), \sigma_2)$$
$$\pi' = \text{update}(\pi, \text{pids}(prog_1), \pi_1, \text{pids}(prog_2), \pi_2)$$

From the induction hypothesis, the following postconditions must also hold:

$$Q_1[\sigma_1/\sigma, \pi_1/\pi, t_1{}^\frown\text{visibleTrace}(t_1')/h] \ \wedge\ Q_2[\sigma_2/\sigma, \pi_2/\pi, t_2{}^\frown\text{visibleTrace}(t_2')/h]$$

Since both postconditions are framed to their respective subprograms, they also hold for (σ', π') instead of (σ, π). It only remains to be proven that $h{}^\frown\text{visibleTrace}(t) \in (t_1{}^\frown\text{visibleTrace}(t_1') \parallel t_2{}^\frown\text{visibleTrace}(t_2'))$, which can be achieved with the lemmas 4.2 and 4.3.

□

5 Examples of Usage

In order to show the applicability of the proof calculus, we provide three examples. The first one is an implementation of the Fibonacci function, showing the verification of sequential processes. The second example is a generic "worker process" which runs a given computation over the values received through a channel, sending the results through another one. The third example is the synchronization between the worker process and a simple client that sends a single value to be analyzed, then obtains the result. All examples presented in this section were formalized in Isabelle.

In order to effectively reason about concrete examples, however, we must have concrete instantiations of LLDo's instruction schemes. We therefore begin by providing these concrete instructions, as well as important lemmas that are used in the examples that follow.

5.1 Preliminary Definitions

Since the definition of LLDo and its instruction schemes are very generic, the definition of concrete instantiations of the schemes is very useful when dealing with concrete examples. Furthermore, LLDo is generic regarding the type of values contained in the registers. For all examples, we postulate that registers contain only integer values.

The following definitions instantiate the instruction schemes to resemble instructions of usual unstructured languages. The members of their $\mathrm{regs_i}$ sets may be computed by syntactical analysis, simply collecting all registers that occur in the instruction. This definition of their $\mathrm{regs_i}$ sets has been verified in Isabelle, as well as their well-formedness. Furthermore, it was proven that $\mathrm{regs_i}(\mathtt{br}\, l) = \emptyset$, and therefore this instruction is well-formed.

- Jump to l when the value of r is zero:
  $\mathtt{bz}\, r \stackrel{def}{=} \mathtt{cbr}\, (\lambda\, \sigma.\ \sigma(r) = 0)\, l$
- Jump to l when the value of r is negative:
  $\mathtt{bn}\, r \stackrel{def}{=} \mathtt{cbr}\, (\lambda\, \sigma.\ \sigma(r) < 0)\, l$
- Jump to l when the value of the two registers is the same:
  $\mathtt{beq}\, r_1\, r_2 \stackrel{def}{=} \mathtt{cbr}\, (\lambda\, \sigma.\ \sigma(r_1) = \sigma(r_2))\, l$
- Jump to l when the value of the first register is less that the value of the second:
  $\mathtt{blt}\, r_1\, r_2 \stackrel{def}{=} \mathtt{cbr}\, (\lambda\, \sigma.\ \sigma(r_1) < \sigma(r_2))\, l$

- Store the constant value v in the register:
  $\texttt{loadi}\ v\ r \stackrel{def}{=} \texttt{do}\ (\lambda\,\sigma.\ \{\sigma[r := v]\})$
- Increment the value of the register:
  $\texttt{inc}\ r \stackrel{def}{=} \texttt{do}\ (\lambda\,\sigma.\ \{\sigma[r := \sigma(r) + 1]\})$
- Store the value of the second register in the first one:
  $\texttt{copy}\ r_d\ r \stackrel{def}{=} \texttt{do}\ (\lambda\,\sigma.\ \{\sigma[r_d := \sigma(r)]\})$
- Add the value of the two registers, storing the result in another:
  $\texttt{add}\ r_d\ r_1\ r_2 \stackrel{def}{=} \texttt{do}\ (\lambda\,\sigma.\ \{\sigma[r_d := \sigma(r_1) + \sigma(r_2)]\})$
- Subtract the value of the two registers, storing the result in another:
  $\texttt{sub}\ r_d\ r_1\ r_2 \stackrel{def}{=} \texttt{do}\ (\lambda\,\sigma.\ \{\sigma[r_d := \sigma(r_1) - \sigma(r_2)]\})$
- Multiply the value of the two registers, storing the result in another:
  $\texttt{mul}\ r_d\ r_1\ r_2 \stackrel{def}{=} \texttt{do}\ (\lambda\,\sigma.\ \{\sigma[r_d := \sigma(r_1) \cdot \sigma(r_2)]\})$
- Send the value of the register through the given channel:
  $\texttt{sendr}\ r\ c \stackrel{def}{=} \texttt{send}\ (\lambda\,\sigma.\ \sigma(r))\ c$
- Receive a value through the given channel and store it in the register:
  $\texttt{recvr}\ r\ c \stackrel{def}{=} \texttt{recv}\ (\lambda\,v\,\sigma.\ \sigma[r := v])\ c$

5.2 Fibonacci

In this subsection, a program computing the Fibonacci function is verified, indicating that the correctness calculus for LLDo is capable of proving the correctness of sequential processes. In particular, it is an example of how a loop may be expressed with unstructured code, and how its correctness may be verified using a loop invariant. The LLDo program for calculating the Fibonacci function may be seen in Figure 5.1.

The specification of the program may be seen in Figure 5.2, reflecting the expected behaviour of the program: if as it starts execution the register r_n contains the natural number n, then as it stops the register r_c will contain $\text{fib}(n)$. Furthermore, the trace is not modified. Instead of simply specifying that the trace remains the same, we state something stronger: any predicate that holds for the initial trace will also hold for the final trace. In particular, if for some trace t we have $h = t$ before execution, the same will hold after the execution — the trace remained the same. We also use a function $n : \text{int trace} \longrightarrow \mathbb{N}$ instead of a simple natural

$$
\begin{array}{l}
\mathit{fibonacci}(l) \stackrel{def}{=} \\
\qquad\begin{array}{rll}
l & ::_s \texttt{loadi}\ 1\ r_i & \oplus \\
1+l & ::_s \texttt{loadi}\ 1\ r_c & \oplus \\
2+l & ::_s \texttt{loadi}\ 1\ r_p & \oplus \\
3+l & ::_s \texttt{br}\ (8+l) & \oplus \\
4+l & ::_s \texttt{copy}\ r_t\ r_c & \oplus \\
5+l & ::_s \texttt{add}\ r_c\ r_c\ r_p & \oplus \\
6+l & ::_s \texttt{copy}\ r_p\ r_t & \oplus \\
7+l & ::_s \texttt{inc}\ r_i & \oplus \\
8+l & ::_s \texttt{blt}\ r_i\ r_n\ (4+l) &
\end{array}
\end{array}
$$

Figure 5.1: Implementation of the Fibonacci function in LLDo. The parameter is received in register r_n, the result is given in r_c. The registers are assumed distinct.

number. This allows us to reuse the proof of correctness in a context where the initial value of r_n depends on previous communication.

$$
\begin{array}{l}
\{\sigma(r_n) = n(h) \wedge n(h) \in \mathbb{N} \wedge l = l_0 \wedge P(h)\} \\
\quad \mathit{fibonacci}(l_0) \\
\{\sigma(r_c) = \mathrm{fib}(n(h)) \wedge n(h) \in \mathbb{N} \wedge l = l_0 + 9 \wedge P(h)\}
\end{array}
$$

$$
\mathrm{fib}(n) = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ \mathit{fib}(n-1) + \mathit{fib}(n-2) & \text{if } n > 1 \end{cases}
$$

Figure 5.2: Specification of the Fibonacci implementation.

In order for us to verify the correctness of this implementation, it is useful to define the expected preconditions P_l for each of the instructions $l ::_s \mathit{instr}$ of the program. These will also often be used as postconditions during the verification. Furthermore, we denote the postcondition as P_9. These assertions are defined in Figure 5.3. We also denote disjunctions like $I_1 \vee I_2 \vee I_9$ by I_{129}, and the part of the fibonacci program whose instructions are defined with labels l such that $l_1 + l_0 \leq l \leq l_2 + l_0$ by $\mathit{fibonacci}_{l_1 - l_2}(l_0)$.

For proving that the specified triple is valid, we will divide the program in three blocks: the initialization, from l to $l+3$; the loop body, from $l+4$ to $l+7$; and the loop test, at $l+8$. These blocks are chosen so that they have a single point of entry — the first instruction — and a single point of exit — the last

$$
\begin{aligned}
I &\equiv \sigma(r_n) = n(h) \wedge n(h) \in \mathbb{N} \wedge P(h) \\
P_0 &\equiv I \wedge pc = l_0 \\
P_1 &\equiv I \wedge pc = 1 + l_0 \wedge \sigma(r_i) = 1 \\
P_2 &\equiv I \wedge pc = 2 + l_0 \wedge \sigma(r_i) = 1 \wedge \sigma(r_c) = 1 \\
P_3 &\equiv I \wedge pc = 3 + l_0 \wedge \sigma(r_i) = 1 \wedge \sigma(r_c) = 1 \wedge \sigma(r_p) = 1 \\
P_4 &\equiv I \wedge 0 < \sigma(r_i) < \sigma(r_n) \wedge pc = 4 + l \\
&\quad \wedge \sigma(r_c) = \mathrm{fib}(\sigma(r_i)) \wedge \sigma(r_p) = \mathrm{fib}(\sigma(r_i) - 1) \\
P_5 &\equiv I \wedge 0 < \sigma(r_i) < \sigma(r_n) \wedge pc = 5 + l \\
&\quad \wedge \sigma(r_c) = \mathrm{fib}(\sigma(r_i)) \wedge \sigma(r_p) = \mathrm{fib}(\sigma(r_i) - 1) \wedge \sigma(r_t) = \mathrm{fib}(\sigma(r_i)) \\
P_6 &\equiv I \wedge 0 < \sigma(r_i) < \sigma(r_n) \wedge pc = 6 + l \\
&\quad \wedge \sigma(r_c) = \mathrm{fib}(\sigma(r_i) + 1) \wedge \sigma(r_p) = \mathrm{fib}(\sigma(r_i) - 1) \wedge \sigma(r_t) = \mathrm{fib}(\sigma(r_i)) \\
P_7 &\equiv I \wedge 0 < \sigma(r_i) < \sigma(r_n) \wedge pc = 7 + l \\
&\quad \wedge \sigma(r_c) = \mathrm{fib}(\sigma(r_i) + 1) \wedge \sigma(r_p) = \mathrm{fib}(\sigma(r_i)) \\
P_8 &\equiv I \wedge pc = 8 + l_0 \\
&\quad \wedge ((\sigma(r_n) = 0 \wedge \sigma(r_i) = 1) \vee (\sigma(r_i) \leq \sigma(r_n) \wedge \sigma(r_i) > 0)) \\
&\quad \wedge \sigma(r_c) = \mathrm{fib}(\sigma(r_i)) \wedge \sigma(r_p) = \mathrm{fib}(\sigma(r_i) - 1) \\
P_9 &\equiv n(h) \in \mathbb{N} \wedge P(h) \wedge pc = 9 + l_0 \wedge \sigma(r_c) = \mathrm{fib}(\sigma(r_n))
\end{aligned}
$$

Figure 5.3: Assertions for verifying the Fibonacci implementation.

instruction —, maintaining a linear flow of execution within them. To prove the correctness within each of these blocks, we follow the pattern shown in Figure 5.4[3]. We essentially use HAPPEND to divide the block into the first instruction and the rest of the block, prove the individual instruction with the appropriate rule then proceed subdividing until we reach the last instruction.

When using HAPPEND, however, the proved triple is based on an invariant. We always choose it to be the disjunction of the block's postcondition and the of preconditions for the first and second instructions of the block. Since the precondition for the first instruction is the precondition of the block itself, and since only the postcondition accepts a PC outside the domain of the block, We can clearly use HCONSEQ to obtain the required triple.

When using the rules for single instructions, on the other hand, it is easiest to

[3]Note that on the figure we omit parts of preconditions when applying HDO, namely those that assume the PC does not point to the instruction. The proofs for these cases are usually trivial.

prove a triple with a single pre- and postcondition. This can also be adapted with HCONSEQ, since the precondition is the only part of the block invariant that accepts the current PC and since the postcondition implies a disjunction that contains it. The only parts left to prove are the premises of the instruction rules, i.e. HDO, which are quite trivial. An example is the proof for the first instruction of the program:

$$
\begin{aligned}
& P_0 \Longrightarrow \forall\, \sigma' \in \{\sigma[r_i := 1]\}.\ P_1[\sigma'/\sigma, pc{+}1/pc] \\
\equiv\ & P_0 \Longrightarrow P_1[\sigma[r_i := 1]\,/\sigma, pc{+}1/pc] \\
\equiv\ & I \wedge pc = l_0 \Longrightarrow I[\sigma[r_i := 1]\,/\sigma, pc{+}1/pc] \wedge pc + 1 = 1 + l_0 \wedge \sigma[r_i := 1](r_i) = 1 \\
\equiv\ & I \Longrightarrow I[\sigma[r_i := 1]/\sigma, pc{+}1/pc] \\
\equiv\ & \sigma(r_n) = n(h) \wedge n(h) \in \mathbb{N} \wedge P(h) \Longrightarrow \sigma[r_i := 1](r_n) = n(h) \wedge n(h) \in \mathbb{N} \wedge P(h) \\
\equiv\ & \sigma(r_n) = n(h) \Longrightarrow \sigma(r_n) = n(h) \\
\equiv\ & \top
\end{aligned}
$$

In order to combine the proofs for the single blocks, we use a similar pattern, which may be seen in Figure 5.5. The implications that need to be proven for the application of HCONSEQ are, again, rather trivial.

5.3 Worker Process

It often occurs, in distributed systems, that an algorithm has to be frequently executed with different inputs. It is the case, for instance, of search engines that must respond to thousands of requests per second. In order to deal with a heavy load, many worker processes are often employed. Each worker is tasked with receiving the inputs for the algorithm, computing the result and sending it back. In this subsection, a generic worker schema is implemented in LLDo. The schema is defined in terms of the implementation $impl : \text{lab} \longrightarrow \text{int seq\_code}$ of a function $f : \mathbb{Z} \longrightarrow \mathbb{Z}$. We assume that the implementation is well-formed and that its instructions occupy d labels. Furthermore, when we begin executing $impl(l_0)$ at label l_0, its execution should terminate at $l_0 + d$.

The implementation of the worker may be seen in Figure 5.6. The worker consists of a loop in which it receives a value and, if it is a negative number, stops. If it received a natural number, the worker proceeds by running the computation over the number and communicating the results. Usually, such worker processes would run indefinitely, but since the proof calculus can only handle terminating executions, the termination upon receival of a negative number was introduced.

The specification of the worker may be seen in Figure 5.7. It basically states that, if started correctly and with a trace that doesn't contain any events on both its

$$\dfrac{\dfrac{pc \in \{l_0\} \wedge P_{018} \Longrightarrow P_0 \qquad P_1 \Longrightarrow pc \notin \{l_0\} \wedge P_{018} \qquad \dfrac{P_0 \Longrightarrow \forall\, \sigma' \in \{\sigma[r_1 := 1]\}.\ P_1[\sigma'/\sigma, pc+1/pc]}{\vdash \{P_0\}\ l_0 ::_s \texttt{loadi}\ 1\ r_i\ \{P_1\}}\ \textsc{Hdo}}{\vdash \{pc \in \{l_0\} \wedge P_{018}\}\ l_0 ::_s \texttt{loadi}\ 1\ r_i\ \{pc \notin \{l_0\} \wedge P_{018}\}}\ \textsc{Hconseq} \qquad \dfrac{\dfrac{\vdots}{\begin{array}{c}\{pc \in \{l_0\} \wedge P_{018}\} \\ \vdash\ l_0 ::_s \texttt{loadi}\ 1\ r_i \\ \{pc \notin \{l_0\} \wedge P_{018}\}\end{array}} \qquad \dfrac{\vdots}{\begin{array}{c}\{pc \in \{l_0{+}1 ... l_0{+}3\} \wedge P_{018}\} \\ \vdash\ \mathit{fibonacci}_{1-3}(l_0) \\ \{pc \notin \{l_0{+}1 ... l_0{+}3\} \wedge P_{018}\}\end{array}}}{\vdash \{P_{018}\}\ \mathit{fibonacci}_{0-3}(l_0)\ \{pc \notin \{l_0 ... l_0{+}3\} \wedge P_{018}\}}\ \textsc{Happend}}{\vdash \{P_0\}\ \mathit{Fibonacci}_{0-3}(l_0)\ \{P_8\}}\ \textsc{Hconseq}$$

Figure 5.4: Simplified pattern of the proof tree for the verification of single blocks within *fibonacci*.

$$
\dfrac{\dfrac{\dfrac{\vdots}{\vdash \{P_0\}\ \mathit{fibonacci}_{0-4}(l_0)\ \{P_8\}}}{\dfrac{\{l \in \{l_0...4{+}l_0\}\} \wedge P_{089}\}}{\vdash \quad \mathit{fibonacci}_{0-4}(l_0)}}{\{l \notin \{l_0...4{+}l_0\}\} \wedge P_{089}\}}\ \textsc{Hconseq} \qquad \dfrac{\vdots}{\dfrac{\{l \in \{5{+}l_0...8{+}l_0\}\} \wedge P_{089}\}}{\vdash \quad \mathit{fibonacci}_{5-8}(l_0)}}{\{l \notin \{5{+}l_0...8{+}l_0\}\} \wedge P_{089}\}}\ \textsc{Hconseq}}{\vdash \{P_{089}\}\,\mathit{fibonacci}(l_0)\,\{l \notin \{0..8\} \wedge P_{089}\}}\ \textsc{Happend}
$$

$$
\dfrac{P_0 \Longrightarrow P_{089} \qquad l \notin \{l_0..8{+}l_0\} \wedge P_{089} \Longrightarrow P_9 \qquad \dfrac{\vdots}{\vdash \{P_{089}\}\,\mathit{fibonacci}(l_0)\,\{l \notin \{0..8\} \wedge P_{089}\}}\ \textsc{Happend}}{\vdash \{P_0\}\ \mathit{fibonacci}(l_0)\ \{P_9\}}\ \textsc{Hconseq}
$$

Figure 5.5: Pattern of the proof tree for the combination of the blocks within *fibonacci*.

$$
\begin{aligned}
worker(l) \stackrel{def}{=} & \\
& \begin{array}{rll}
l & ::_s \texttt{recv}\, r_{in}\, c_{in} & \oplus \\
1+l & ::_s \texttt{bn}\, r_{in}\, (4+d+l) & \oplus \\
 & impl(2+l) & \oplus \\
2+d+l & ::_s \texttt{send}\, r_{out}\, c_{out} & \oplus \\
3+d+l & ::_s \texttt{br}\, l & \oplus \\
4+d+l & ::_s \texttt{send}\, r_{in}\, c_{out} &
\end{array}
\end{aligned}
$$

Figure 5.6: Implementation of a worker process in LLDo. Values are received through channel c_{in}, results are sent through c_{out}. As soon as a negative value is received, the worker stops and sends the same value through c_{out}.

channels, the resulting trace contains a series of receiving events of natural numbers on c_{in}, followed by a last receiving event of a negative number. It also contains a series of sending events of the results on c_{out}, followed by a last sending event of the received negative number. Furthermore, the number of events with c_{in} and with c_{out} is the same.

$$
\begin{aligned}
&\forall\, l_0\, P\, n.\ \vdash \{pc = l_0 \wedge P(h) \wedge \sigma(r_{in}) = n(h) \wedge n(h) \in \mathbb{N}\} \\
&\qquad\qquad impl(l_0) \\
&\qquad\qquad \{pc = d + l_0 \wedge P(h) \wedge \sigma(r_{out}) = f(n(h)) \wedge n(h) \in \mathbb{N}\} \Longrightarrow \\
&\vdash \{pc = l_0 \wedge h \downarrow_{c_{in}\, c_{out}} = \langle\rangle\} \\
&\quad impl(2+l) \\
&\quad \left\{ \begin{array}{l}
\quad pc = 5 + d + l_0 \wedge \text{length}(h \downarrow_{c_{in}}) = \text{length}(h \downarrow_{c_{out}}) \\
\wedge\ (\forall\, i < \text{length}(h \downarrow_{c_{in}}) - 1.\ \exists\, n \in \mathbb{N}.(h \downarrow_{c_{in}})[i] = c_{in}?n) \\
\wedge\ (\forall\, i < \text{length}(h \downarrow_{c_{out}}) - 1.\ (h \downarrow_{c_{out}})[i] = c_{out}?f(\text{val}(c_{in}[i]))) \\
\wedge\ (\exists\, n.n < 0 \wedge \text{last}(h \downarrow_{c_{in}}) = c_{in}?n \wedge last(h \downarrow_{c_{out}}) = c_{out}!n)
\end{array} \right\}
\end{aligned}
$$

Figure 5.7: Specification of the worker process.

The proof that the specification of the worker holds is done with the same approach as in the fibonacci example. The definitions of the preconditions P_i for each instruction in label $i + l_0$, as well as of the postcondition P_{5+d}, can be found in

Figure 5.8.

With this example we can see the advantages of a composable proof calculus: the actual algorithm executed by the worker may be implemented and verified separately. We may, in fact, use the implementation of the fibonacci function, and its specification we already proved correct suffices for the correctness of the worker.

5.4 Simple Client and Synchronization

Although the last two examples only involved single sequential processes, the proof calculus may also be employed for the verification of concurrent programs. For that, we specify a simple client for the worker process. This client sends a single value to be processed and, after receiving the result, sends -1 to terminate the execution of the worker. The specification of the combined program may be seen in Figure 5.9.

The proof that the program is correct follows by straightforward application of the rules HCONSEQ, HPAR and HONE. The correctness of the individual processes, necessary for HONE, is obtained from the assumption of the lemma and from the verification of the worker. The framing of the assertions, necessary for HPAR, is trivial to verify. The important parts of the proof are the following two lemmas.

Lemma 5.1.

$$\pi(0) = 0 \land \pi(1) = 0 \land t = \langle\rangle \Longrightarrow$$
$$\exists\, t_1\, t_2 . t \in (t_1 \parallel t_2) \land (\pi(0) = 0 \land t_1 = \langle\rangle) \land (\pi(1) = 0 \land t_2 \lfloor_{c_{in}\, c_{out}} = \langle\rangle)$$

Proof. We take $t_1 = t_2 = \langle\rangle$. The equalities of the PCs and traces are trivially verified. The synchronization of the traces is provided by SYNCSEMPTY.

□

$$
\begin{aligned}
P_0 &\equiv pc = l_0 \wedge \mathrm{length}(h \restriction_{c_{in}}) = \mathrm{length}(h \restriction_{c_{out}}) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{in}}).\ \exists\, n \in \mathbb{N}.\ (h \restriction_{c_{in}})[i] = c_{in}?n) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{out}}).\ (h \restriction_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \\
P_1 &\equiv pc = 1 + l_0 \wedge \mathrm{length}(h \restriction_{c_{in}}) = 1 + \mathrm{length}(h \restriction_{c_{out}}) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{in}} - 1).\ \exists\, n \in \mathbb{N}.\ (h \restriction_{c_{in}})[i] = c_{in}?n) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{out}}).\ (h \restriction_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \\
&\quad \wedge (\exists\, n.\ \mathrm{last}(h \restriction_{c_{in}}) = c_{in}?n) \wedge \sigma(r_{in}) = \mathrm{val}(\mathrm{last}(h \restriction_{c_{in}})) \\
P_2 &\equiv pc = 2 + l_0 \wedge \mathrm{length}(h \restriction_{c_{in}}) = 1 + \mathrm{length}(h \restriction_{c_{out}}) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{in}}).\ \exists\, n \in \mathbb{N}.\ (h \restriction_{c_{in}})[i] = c_{in}?n) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{out}}).\ (h \restriction_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \\
&\quad \wedge \sigma(r_{in}) = \mathrm{val}(\mathrm{last}(h \restriction_{c_{in}})) \\
P_{2+d} &\equiv pc = 2 + d + l_0 \wedge \mathrm{length}(h \restriction_{c_{in}}) = 1 + \mathrm{length}(h \restriction_{c_{out}}) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{in}}).\ \exists\, n \in \mathbb{N}.\ (h \restriction_{c_{in}})[i] = c_{in}?n) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{out}}).\ (h \restriction_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \\
&\quad \wedge \sigma(r_{out}) = f(\mathrm{val}(\mathrm{last}(h \restriction_{c_{in}}))) \\
P_{3+d} &\equiv pc = 3 + d + l_0 \wedge \mathrm{length}(h \restriction_{c_{in}}) = 1 + \mathrm{length}(h \restriction_{c_{out}}) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{in}} - 1).\ \exists\, n \in \mathbb{N}.\ (h \restriction_{c_{in}})[i] = c_{in}?n) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{out}}).\ (h \restriction_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \\
&\quad \wedge (\exists\, n.\ n < 0 \wedge \mathrm{last}(h \restriction_{c_{in}}) = c_{in}?n) \wedge \sigma(r_{in}) = \mathrm{val}(\mathrm{last}(h \restriction_{c_{in}})) \\
P_{4+d} &\equiv pc = 4 + d + l_0 \wedge \mathrm{length}(h \restriction_{c_{in}}) = 1 + \mathrm{length}(h \restriction_{c_{out}}) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{in}} - 1).\ \exists\, n \in \mathbb{N}.\ (h \restriction_{c_{in}})[i] = c_{in}?n) \\
&\quad \wedge (\forall\, i < \mathrm{length}(h \restriction_{c_{out}} - 1).\ (h \restriction_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \\
&\quad \wedge (\exists\, n.\ n < 0 \wedge \mathrm{last}(h \restriction_{c_{in}}) = c_{in}?n \wedge \mathrm{last}(h \restriction_{c_{out}}) = c_{out}?n)
\end{aligned}
$$

Figure 5.8: Assertions for verifying the worker implementation.

$$
\begin{array}{l}
\vdash \{pc = 0 \wedge h = \langle\rangle\} \\
\quad \mathit{client} \\
\quad \{(\exists\, n \in \mathbb{N}.\ h \lfloor_{c_{in}} = \langle c_{in}!n, c_{in}! - 1\rangle) \wedge (\exists\, n_1\ n_2.\ h \lfloor_{c_{out}} = \langle c_{out}!n_1, c_{out}!n_2\rangle\} \Longrightarrow \\
\vdash \{\pi(0) = 0 \wedge \pi(1) = 0 \wedge h \lfloor_{c_{in}\, c_{out}} = \langle\rangle\} \\
\quad 0 ::_c \mathit{client} \parallel 1 ::_c \mathit{worker}(0) \\
\quad \{\mathrm{fullySynced}(t) \Longrightarrow \exists\, n \in \mathbb{N}.\ h \lfloor_{c_{in}} = \langle c_{in}.n, c_{in}. - 1\rangle \wedge h \lfloor_{c_{out}} = \langle c_{out}.f(n), c_{out}. - 1\rangle\}
\end{array}
$$

Figure 5.9: Specification of the combined worker and client.

Lemma 5.2.

$$
\begin{array}{l}
t \in (t_1 \parallel t_2) \wedge \\
(\exists\, n \in \mathbb{N}.\ t_1 \lfloor_{c_{in}} = \langle c_{in}!n, c_{in}! - 1\rangle) \wedge \\
(\exists\, n_1\ n_2.\ t_1 \lfloor_{c_{out}} = \langle c_{out}!n_1, c_{out}!n_2\rangle \wedge \\
\mathrm{length}(t_2 \lfloor_{c_{in}}) = \mathrm{length}(t_2 \lfloor_{c_{out}}) \wedge \\
(\forall\, i < \mathrm{length}(t_2 \lfloor_{c_{in}}) - 1.\ \exists\, n \in \mathbb{N}.(t_2 \lfloor_{c_{in}})[i] = c_{in}?n) \wedge \\
(\forall\, i < \mathrm{length}(t_2 \lfloor_{c_{out}}) - 1.\ (t_2 \lfloor_{c_{out}})[i] = c_{out}?f(\mathrm{val}(c_{in}[i]))) \wedge \\
(\exists\, n.n < 0 \wedge \mathrm{last}(t_2 \lfloor_{c_{in}}) = c_{in}?n \wedge \mathit{last}(t_2 \lfloor_{c_{out}}) = c_{out}!n) \wedge \\
\mathrm{fullySynced}(t) \Longrightarrow \\
(\exists\, n \in \mathbb{N}.\ t \lfloor_{c_{in}} = \langle c_{in}.n, c_{in}. - 1\rangle \wedge t \lfloor_{c_{out}} = \langle c_{out}!f(n), c_{out}. - 1\rangle)
\end{array}
$$

Proof. We begin by proving that $t \lfloor_{c_{in}} = \langle c_{in}.n, c_{in}.-1\rangle \wedge t_2 \lfloor_{c_{in}} = \langle c_{in}?n, c_{in}?-1\rangle$. We note that $t \lfloor c_{in} \in (t_1 \lfloor c_{in} \parallel t_2 \lfloor c_{in})$ (lemma **??**). The last step of the derivation of this fact must have been an application of SYNCSsync — SYNCSempty cannot be applied because $t_1 \lfloor c_{in}$ is not empty, SYNCSleft and SYNCSright because any events coming from t_1 or t_2 are unsynchronized and would contradicting the assumption that fullySynced(t).

Therefore, we have $t \lfloor_{c_{in}} = \langle e\rangle^\frown t'$, with $e = c_{in}!n \parallel c_{in}?n = c_{in}.n$. The following also hold.

$$
\begin{array}{c}
t_1 = \langle c_{in}!n\rangle^\frown t_1' \wedge t_2 = \langle c_{in}!n\rangle^\frown t_2' \\
t' \in (t_1' \parallel t_2')
\end{array}
$$

We proceed with the same kind of case analysis on the derivation of the synchronization of the subtraces, obtaining the following.

$$t' = \langle c_i n. -1 \rangle^\frown t'' \wedge t_1 = \langle c_{in}! -1 \rangle^\frown \langle\rangle \wedge t_2 = \langle c_{in}! -1 \rangle^\frown t_2''$$
$$t'' \in (\langle\rangle \parallel t_2'')$$

The only way have to derived this synchronization of traces is with SYNCSempty, therefore $t'' = t_2'' = \langle\rangle$. We thus have the following, and analogous equalities for $t_2 \upharpoonright_{c_{in}}$.

$$\begin{aligned} t \upharpoonright_{c_{in}} &= \langle c_{in}.n \rangle^\frown t' \\ &= \langle c_{in}.n \rangle^\frown \langle c_i n. -1 \rangle^\frown \langle\rangle \\ &= \langle c_{in}.n, c_{in}. -1 \rangle \end{aligned}$$

The remaining goal of $t \upharpoonright_{c_{out}} = \langle c_{out}! f(n), c_{out}. -1 \rangle$ can be shown by an analogous sequence of case analysis.

□

6 Related Work

The semantics of sequential unstructured languages have already been explored. Myreen [10] defined a similar but more detailed framework than the one used as a basis for this thesis. This framework is defined independently of a machine model, allowing multiple real architectures to be modelled and reasoned about.

Another active research area deals with non-functional properties of unstructured code. In [2], for example, the timing behaviour of LLVM IR[4] is modelled by the given operational semantics.

Concurrency has also been studied in the context of low-level languages, although with a different focus than this thesis. Several features of modern processors — such as out-of-order execution and independent caches for multiple cores — highly impact the behaviour of concurrent programs with shared variables. Therefore, the current approaches are usually tied to specific architectures. Operational semantics for the POWER and ARM architectures are defined in [1], and for Intel's x86 in [15].

[4] An assembly-like intermediate language; part of the LLVM framework, which is used by many compilers.

7 Conclusions

In this thesis, the operational semantics and proof calculus for the Low-Level Do (LLDo) language was provided. A small-step presentation of the semantics was provided to ensure the formal specification matches the informal, intuitive understanding of the language. Existing techniques for compositionally dealing with either unstructured or communicating code were combined on the big-step presentation of the semantics, which was proven equivalent to the small-step presentation for terminating executions[5].

Furthermore, a proof calculus for verifying the correctness of LLDo programs was also defined. The calculus is also compositional and was proven correct with respect to the big-step semantics. The applicability of the calculus for verifying single processes was shown through three examples: an implementation of the fibonacci function; a generic "worker process" which executes some specific computation over the values it receives through a channel, which is a common pattern in distributed systems; a simple client for this worker, which requests the computation of a single value. The advantages of the composability were shown by instantiating the worker with the implementation of the fibonacci and proving its correctness by reusing the previous proofs. The applicability of the calculus for verifying the synchronization of concurrent programs was shown by proving the correctness of the syncronization between a worker process and the simple client.

Since the proof calculus provided in this thesis only deals with terminating programs, it cannot be applied non-terminating, so-called *reactive* systems. A possible future extension is therefore dealing with specifications that, besides pre- and post-conditions, contain *invariants*. This approach for verifying non-terminating programs was already explained by Hooman [7] in the context of structured code. Another compositional presentation of the formal semantics would have to be developed, since the big-step semantics only specify terminating executions of programs.

Since non-functional properties such as timing are also important in many classes of safety-critical systems, such as real-time systems, this is another possible extension. In particular, the approach of Bartels and Glesner [2] could be extended to work with communicating, unstructured code in a compositional way.

[5]The big-step semantics can only specify the behaviour of terminating executions, while the small-step semantics may also express unfinished executions.

References

[1] Alglave, J., Fox, A., Ishtiaq, S., Myreen, M. O., Sarkar, S., Sewell, P., and Nardelli, F. Z. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming* (2009), ACM, pp. 13–24.

[2] Bartels, B., and Glesner, S. Verification of distributed embedded real-time systems and their low-level implementations using timed csp. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific* (2011), IEEE, pp. 195–202.

[3] Bartels, B., and Jähnig, N. Mechanized, compositional verification of low-level code. In *NASA Formal Methods*, J. Badger and K. Rozier, Eds., vol. 8430 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 98–112.

[4] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (1969), 576–580.

[5] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[6] Hoare, C. A. R. *Communicating sequential processes*. Prentice-Hall, 1985.

[7] Hooman, J., de Roever, W.-P., Pandya, P., Xu, Q., Zhou, P., and Schepers, H. *A Compositional Approach to Concurrency and its Applications*. 2003.

[8] Milner, R. A calculus of communicating systems.

[9] Milner, R. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[10] Myreen, M. O. *Formal verification of machine-code programs*. University of Cambridge, Computer Laboratory, Trinity College, 2008.

[11] Necula, G. *Proof-carrying code*. Springer, 2011.

[12] Pierce, B. C. *Types and programming languages*. MIT press, 2002.

[13] Roscoe, A. W., Hoare, C. A., and Bird, R. *The theory and practice of concurrency*, vol. 169. Prentice Hall Englewood Cliffs, 1998.

[14] Saabas, A., and Uustalu, T. A compositional natural semantics and hoare logic for low-level languages. *Theoretical Computer Science 373*, 3 (2007), 273–302.

[15] Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z., and Myreen, M. O. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53*, 7 (2010), 89–97.