UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CÁLCULO DE AMBIENTES TIPADO SENSÍVEL AO CONTEXTO PARA APLICAÇÕES PERVASIVAS

DISSERTAÇÃO DE MESTRADO

Douglas Pereira Pasqualin

Santa Maria, RS, Brasil

2012

CÁLCULO DE AMBIENTES TIPADO SENSÍVEL AO CONTEXTO PARA APLICAÇÕES PERVASIVAS

Douglas Pereira Pasqualin

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-Graduação em Informática (PPGI), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Profª. Drª. Juliana Kaizer Vizzotto

Santa Maria, RS, Brasil

2012

Pasqualin, Douglas Pereira

Cálculo de Ambientes Tipado Sensível ao Contexto para Aplicações Pervasivas / por Douglas Pereira Pasqualin. – 2012.
105 f.: il.; 30 cm.

Orientadora: Juliana Kaizer Vizzotto
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2012.

1. Sistema de Tipos. 2. Cálculo de Ambientes. 3. Computação Pervasiva. 4. Sensibilidade ao Contexto. I. Vizzotto, Juliana Kaizer. II. Título.

E-mail: douglas.pasqualin@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

CÁLCULO DE AMBIENTES TIPADO SENSÍVEL AO CONTEXTO PARA APLICAÇÕES PERVASIVAS

elaborada por
Douglas Pereira Pasqualin

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Juliana Kaizer Vizzotto, Dr[a].
(Presidente/Orientadora)

Álvaro Freitas Moreira, Dr. (UFRGS)

Eduardo Kessler Piveta, Dr. (UFSM)

Santa Maria, 25 de Maio de 2012.

Dedico este trabalho ao meu filho Nicolas,
que apesar dos seus poucos meses de vida,
consegue me passar confiança, felicidade e serenidade.
Tudo isso, apenas com o seu lindo e sincero sorriso . . .

AGRADECIMENTOS

Durante a minha graduação e vida profissional, sempre trabalhei com a parte prática da computação: a programação em linguagens de alto nível. Foi então, que a professora Juliana me apresentou o outro lado da computação: a teórica. Agradeço a minha orientadora pela confiança, e por ter aceitado me orientar, mesmo sabendo que para desenvolver este trabalho eu precisaria de muita ajuda e estudo. Foram dois anos difíceis, de muito estudo e aprendizado, mas que valeram a pena. Foi um trabalho bastante desafiador, mas como me disse a Juliana: "se fosse fácil, não teria graça!".

Agradeço à minha esposa Daiane, a grande incentivadora deste trabalho, por nunca ter deixado eu desistir, e aguentado meu mau humor, quando tudo parecia que ia dar errado. Sem o seu incentivo constante eu não teria iniciado nem a graduação.

Ao professor e amigo Guilherme Dhein, pela carta de recomendação e por ter me apoiado em momentos críticos, inclusive quando quase desisti de tudo.

Ao professor, e agora também escritor, André Cordenonsi, pela carta de recomendação, e pela indicação da professora Juliana como uma possível orientadora.

À comissão examinadora, Dr. Álvaro Moreira, pelos comentários e sugestões de melhoria no trabalho, e Dr. Eduardo Piveta, pela revisão minuciosa da parte escrita.

À AVMB, por ter flexibilizado meu horário de trabalho durante o primeiro ano do mestrado, permitindo a minha ida às aulas durante o horário de expediente.

Ao Dr. Andrew D. Gordon (*Microsoft Research*), por ter gentilmente me cedido as suas macros pessoais LaTeX, que auxiliaram na tarefa de formatação deste trabalho.

For fim, agradeço à UFSM, pelo ensino gratuito e de qualidade.

“Safe languages can be defined as ones that make it impossible to shoot yourself in the foot while programming.”

(BENJAMIN C. PIERCE)

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

CÁLCULO DE AMBIENTES TIPADO SENSÍVEL AO CONTEXTO PARA APLICAÇÕES PERVASIVAS

AUTOR: DOUGLAS PEREIRA PASQUALIN
ORIENTADORA: JULIANA KAIZER VIZZOTTO
Local da Defesa e Data: Santa Maria, 25 de Maio de 2012.

Atualmente, a computação móvel está mais presente na rotina das pessoas. Celulares, *notebooks*, *smartphones* e redes sem fio fazem parte do cotidiano. Com essa tecnologia disponível, as pesquisas na área de computação pervasiva crescem a cada dia. A ideia da computação pervasiva surgiu com um artigo escrito por Mark Weiser em 1991, com uma visão pessoal de como seria a computação no século 21. Weiser descreveu que a computação faria parte do cotidiano das pessoas, e estaria acessível em todos os ambientes. Além disso, seria tão natural que passaria a ideia de estar "invisível" no ambiente. Para tornar a computação invisível, as aplicações devem ser pró-ativas, solicitando o mínimo de intervenção do usuário para o seu funcionamento. Um conceito importante que surge na computação pervasiva é a "sensibilidade ao contexto". Contexto é qualquer informação que possa ser utilizada para caracterizar uma entidade. Com base em informações contextuais, as aplicações podem se adaptar dinamicamente aos ambientes nos quais estão inseridas, tornando-se pró-ativas e transmitindo a ideia da invisibilidade. Novas linguagens de programação ou até mesmo novos paradigmas de programação estão sendo desenvolvidos, tentando tornar mais intuitiva a programação de aplicações pervasivas. A maioria dessas linguagens tenta adicionar novas funcionalidades em linguagens já existentes. Porém, alguns autores defendem que deveriam existir novos formalismos que ajudem a modelar as propriedades dos sistemas pervasivos, em especial a sensibilidade ao contexto. A descrição formal de um sistema modelado através de métodos formais pode ser utilizada para demonstrar que algumas propriedades de um sistema estão corretamente modeladas. Nesse sentido, este trabalho estuda um modelo formal que pode servir como base para a especificação de novas linguagens de programação, chamado Cálculo de Ambientes Sensível ao Contexto (CASC), proposto para descrever ambientes móveis e aplicações pervasivas. Outro método formal que é utilizado para especificar linguagens de programação são os sistemas de tipos. Sistemas de tipos ajudam a garantir que um sistema se comporta de acordo com a sua especificação, ou seja, são uma maneira de provar formalmente a ausência de comportamentos indesejados dentro de um sistema. Dessa forma, a principal contribuição deste trabalho é a definição de um sistema de tipos para o CASC com o foco no controle de comunicação entre processos. Como estudo de caso, dois cenários reais foram modelados utilizando o CASC, demonstrando o uso do sistema de tipos desenvolvido. A propriedade *preservation* (ou *subject reduction*) do sistema de tipos foi provada formalmente, demostrando que o sistema de tipos está correto, ou seja, atingindo o objetivo principal do trabalho.

Palavras-chave: Sistema de Tipos. Cálculo de Ambientes. Computação Pervasiva. Sensibilidade ao Contexto.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

TYPED CONTEXT AWARENESS AMBIENT CALCULUS FOR PERVASIVE APPLICATIONS

AUTHOR: DOUGLAS PEREIRA PASQUALIN
ADVISOR: JULIANA KAIZER VIZZOTTO
Defense Place and Date: Santa Maria, May 25th, 2012.

Nowadays, mobile computing is more present in daily life. Mobile phones, notebooks, smart phones and wireless networks are part of everyday life. With this technology available, the research in pervasive computing is growing. The idea of pervasive computing was introduced by Mark Weiser in 1991, with a personal vision of how would be computing in the 21st century. Weiser's idea was that information processing would become part of everyday life, and would be available everywhere. Furthermore, it would be so natural as being "invisible" in the ambient. To make computing invisible, applications must be proactive, asking for a minimum of user intervention for its operation. An important concept that arises with pervasive computing is the "context awareness". Context is any information that can be used to characterize an entity. Based on contextual information, applications can dynamically adapt to the environments in which they operate, becoming proactive and conveying the idea of invisibility. New programming languages or even new paradigms are being developed trying to make more intuitive the programming of pervasive applications. Most of these programming languages attempt to add new features into existing programming languages. However, some authors argue that there must be new formalisms that help to model the properties of pervasive systems, in particular the context awareness. The formal description of a system modeled by formal methods can be used to demonstrate that some properties of the system are correctly modeled. In this sense, this work studies a formal model that can be used as a basis for specifying a new programming language, called Calculus of Context-aware Ambients (CCA), proposed to describe mobile and pervasive applications. Another formal method used in the specification of programming languages are the type systems. Type systems helps to ensure that the system behaves according to the specification, that is, is a way to formally prove the absence of undesirable behavior in a system. Thus, the main contribution of this work is the definition of a type system for the CCA with the focus in the communication between processes. As a case study two real scenarios were modeled using the CCA, demonstrating the use of the type system developed. The *preservation* (or *subject reduction*) property of the type system was formally proved, demonstrating that the type system is correct, i.e., achieving the main purpose of the present work.

Keywords: Type Systems. Ambient Calculus. Pervasive Computing. Context Awareness.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BA	*Boxed Ambients*
CA	Cálculo de Ambientes
CASC	Cálculo de Ambientes Sensível ao Contexto
IP	*Internet Protocol*
TRD	Tipo de registro dependente
POO	Programação Orientação a Objetos
SO	Sistema Operacional
UFSM	Universidade Federal de Santa Maria

SUMÁRIO

1 INTRODUÇÃO

Atualmente, a computação móvel está mais presente no cotidiano das pessoas. Com a produção em larga escala, e consequentemente a preços mais acessíveis, *notebooks*, celulares e *smartphones* são itens comuns no dia a dia. Redes sem fio (*wireless*) também já fazem parte do cotidiano, como em aeroportos, universidades, em casa e até mesmo em ônibus de viagens (RUAS, 2010). Além disso, o acesso à Internet já não é mais exclusivo de computadores. Celulares, televisores e tocadores de *Blu-ray* já conseguem usufruir dos serviços oferecidos na *Web*. Com a tecnologia disponível atualmente, as pesquisas na área de computação pervasiva [1] crescem a cada dia.

A ideia da computação pervasiva não é recente. O termo surgiu com um artigo escrito por Mark Weiser (1991), com uma previsão de como seria a computação no século 21. A ideia de Weiser era tornar o uso da computação natural, como hoje é o uso da eletricidade. Ou seja, a computação faria parte do cotidiano das pessoas, e estaria acessível em todos os ambientes. Para tornar o uso da computação natural, ele defendia que a interação do usuário com as máquinas deveria ser de forma natural, ou até mesmo "invisível". Uma das formas de alcançar esse objetivo é o desenvolvimento de interfaces mais naturais para interação, além do tradicional *mouse* e teclado, e colocar o foco do usuário na tarefa a ser executada, e não em como operar a máquina para executar a tarefa. Para tornar o uso da computação "invisível", as aplicações devem ser pró-ativas no sentido de prever as intenções dos usuários, solicitando o mínimo de intervenção dos mesmos para o seu correto funcionamento.

Um conceito importante que surge na computação pervasiva é a *sensibilidade ao contexto*. Contexto é qualquer informação que possa ser utilizada para caracterizar uma entidade (ABOWD et al., 1999). Um contexto pode ser, por exemplo, a localização da aplicação ou usuário, a temperatura do ambiente, ou a hora atual. Com base nessas informações contextuais, as aplicações podem se adaptar dinamicamente aos ambientes nos quais estão inseridas, tornando-se pró-ativas e transmitindo a ideia da invisibilidade.

Novas linguagens de programação ou até mesmo novos paradigmas de programação estão sendo desenvolvidos, tentando tornar mais intuitiva a programação de aplicações pervasivas (MIN et al., 2007; RARAU; BENTA; CREMENE, 2007; ALY; NADI; HAMDAN, 2008;

[1] Conforme (ARAUJO, 2003) o termo "pervasiva" não existe no vocabulário português. Porém, a grande maioria dos artigos escritos em português utiliza o termo "pervasiva" como tradução para o termo inglês "*pervasive*", para evitar confusão com outros termos emergentes da área.

JUNBIN et al., 2009). A maioria dessas linguagens tenta adicionar novas funcionalidades a linguagens de propósito geral e amplamente utilizadas, tais como Java e C#, ou seja, não são linguagens projetadas especificamente para a computação pervasiva. Atualmente, o paradigma mais utilizado para desenvolvimento de aplicações é o paradigma de orientação a objetos. Porém esse modelo é baseado principalmente em ambientes estáticos, nos quais o contexto é estático e possíveis mudanças no ambiente são previsíveis (YAO et al., 2008). Alguns autores, tais como Kjærgaard e Bunde-Pedersen (2006) e Dapoigny e Barlatier (2009), defendem que deveriam existir novos formalismos que ajudem a modelar as propriedades dos sistemas pervasivos, em especial a sensibilidade ao contexto. Métodos formais podem ser utilizados para descrever em qualquer nível de detalhe, um sistema que será desenvolvido. Essa descrição formal pode ser utilizada para demonstrar que o sistema modelado está precisamente de acordo com a especificação (QIN; ZHENG; XING, 2008). Um modelo formal amplamente utilizado para especificação de linguagens de programação é o Cálculo-λ (PIERCE, 1997) (descrito na Seção 3.2). Este modelo pode ser visto como uma linguagem de programação simples na qual qualquer computação pode ser vista como um objeto matemático, cujas propriedades podem ser investigadas formalmente. Existem vários outros modelos formais com objetivos diferentes, como por exemplo, modelar sistemas concorrentes e distribuídos.

Um modelo formal desenvolvido para modelar sistemas móveis é o Cálculo de Ambientes (CA), (CARDELLI; GORDON, 1998). Um ambiente é um lugar onde processos podem trocar mensagens e onde outros ambientes podem entrar e sair. A ideia principal é modelar hierarquicamente ambientes computacionais. A hierarquia entre ambientes pode ser demonstrada através de um exemplo: considere um ambiente *sala*, que está contido dentro um ambiente *andar* que por sua vez está contido dentro de um ambiente *prédio*, etc. A vantagem é poder mover ambientes inteiros (por exemplo entre dispositivos), juntamente com os seus ambientes filhos, ao invés de objetos individuais. Como a mobilidade de código é um dos requisitos para a computação pervasiva, o CA pode ser considerado um modelo formal interessante para descrever aplicações móveis. Um modelo recente foi desenvolvido para adicionar suporte à sensibilidade ao contexto no CA, chamado de Cálculo de Ambientes Sensível ao Contexto (CASC) (SIEWE; CAU; ZEDAN, 2009).

Outro método formal utilizado para especificar linguagens de programação são os sistemas de tipos. Sistemas de tipos ajudam a garantir que um sistema se comporta de acordo com sua especificação, ou seja, são uma maneira de provar formalmente a ausência de comporta-

mentos indesejados dentro de um sistema. Sua principal função é prevenir a ocorrência de erros durante a execução de um programa (PIERCE, 2002; CARDELLI, 2004).

A comunicação entre ambientes é assíncrona e não direcionada. Dessa forma, este trabalho propõe um sistema de tipos para o CASC com o foco no controle de comunicação entre processos. A ideia principal é adicionar anotações de tipos, em processos e ambientes, garantindo que a comunicação entre processos será executada somente se os tipos do processo emissor e receptor combinarem. O sistema de tipos proposto neste trabalho, é inspirado no trabalho de Cardelli e Gordon (CARDELLI; GORDON, 1999).

1.1 Objetivo principal

Desenvolver um sistema de tipos para o CASC, com o objetivo principal de controlar a comunicação entre os processos, ou seja, garantir que os processos somente receberão mensagens que saibam interpretar. Esse controle é possível, adicionado anotações de tipos em processos, e em ambientes, ou seja, conhecendo previamente o tipo de mensagem que o processo espera. Com a inclusão do sistema de tipos, esta dissertação contribui com a especificação do CASC, permitindo, além de descrever o comportamento de aplicações pervasivas, que a comunicação entre processos seja feita de forma correta. Além disso, o CASC poderá servir como base para o desenvolvimento de novas linguagens de programação.

1.2 Organização do texto

O Capítulo 2 apresenta os principais conceitos da computação pervasiva, bem como os seus atuais desafios. O Capítulo 3 revisa de uma forma mais abrangente os principais conceitos sobre os sistemas de tipos. O objetivo desse capítulo, além de servir como fundamentação teórica para esta dissertação, é servir como fonte de pesquisa para futuros discentes interessados em sistemas de tipos. Dessa forma, o leitor já familiarizado com o tema, pode pular a leitura deste capítulo. O Capítulo 4 descreve o Cálculo de Ambientes, assim como alguns sistemas de tipos propostos para ele. Neste mesmo capítulo são apresentadas algumas variantes do CA, entre elas o CASC, que será utilizado como base principal para este trabalho. O Capítulo 5 apresenta o sistema de tipos para o CASC. Finalmente, o Capítulo 6 apresenta as conclusões e algumas propostas para trabalhos futuros.

2 COMPUTAÇÃO PERVASIVA

Mark Weiser (1991), cientista chefe do Centro de Pesquisa Xerox PARC, escreveu um artigo descrevendo a sua visão de como seria a computação no século 21: *"As tecnologias mais profundas são aquelas que desaparecem. Elas se entrelaçam na vida cotidiana até que sejam indistinguíveis dela"*. A ideia de Weiser era tornar natural o uso da computação, assim como a eletricidade (NIEMELA; LATVAKOSKI, 2004), integrando a computação com o mundo físico (WANT; PERING, 2005). Dessa forma, o uso da computação seria inconsciente, dando a ideia de invisibilidade. Na prática, o conceito de invisibilidade pode ser alcançado, minimizando a distração do usuário (SATYANARAYANAN, 2001). Alguns autores divergem sobre o uso do termo pervasivo e ubíquo. Alguns utilizam os dois como sinônimos, outros utilizam somente o termo pervasivo ou ubíquo (geralmente para uma mesma definição). Conforme Augustin (2004), a computação pervasiva é uma forma de computação móvel e global na qual vários dispositivos estarão interconectados e acompanhando a mobilidade do usuário, de forma que a computação possa estar espalhada no ambiente, além de possuir sensibilidade ao contexto. A computação ubíqua aconteceria quando a computação pervasiva estivesse tão presente no nosso dia a dia e agindo proativamente de forma que transparecesse a ideia de invisibilidade e onipresença. Ou seja, a computação pervasiva seria um pré-requisito para a ubíqua. Muitos autores tais como (GRIMM et al., 2001), (HENRICKSEN; INDULSKA, 2004), (YAO et al., 2008), (ALY; NADI; HAMDAN, 2008) e (JUNBIN et al., 2009) utilizam o termo computação pervasiva para a computação que é sensível ao contexto. Por essa razão, esse é o termo que será utilizado neste trabalho para definir a computação sensível ao contexto.

Atualmente, tecnologia necessária para a computação pervasiva já faz parte do cotidiano: celulares, notebooks, *smartphones*, redes sem fio (*wireless*), acesso a Internet através de celulares, etc. Apesar de os dispositivos fazerem parte do cotidiano, Hofer *et al.* (2003) afirma que ainda existem muitos desafios a serem superados para o desenvolvimento de aplicações pervasivas. Um dos principais desafios é a interoperabilidade entre os dispositivos (HUANG; MANGS, 2008). Diferentes dispositivos possuem diferentes funcionalidades, velocidade de processamento e capacidade de armazenamento. Dessa forma, não é possível garantir que um aplicativo desenvolvido para um dispositivo executará em um outro diferente.

A percepção do ambiente é uma das maiores diferenças entre a computação pervasiva e a computação tradicional (SAHA; MUKHERJEE, 2003). As aplicações pervasivas devem ser

pró-ativas no sentido de prever as intenções do usuário, ou seja, a interação com o sistema deve ser natural, como se a tecnologia fizesse parte do ambiente, devendo ser evitadas ao máximo as intervenções do usuário para o funcionamento do sistema. Dispositivos que obtém informações sobre o ambiente no qual estão inseridos, como, por exemplo, a localização do usuário, podem ser utilizados para permitir uma interação mais natural do usuário com o sistema.

As próximas seções descrevem outras características da computação pervasiva. A Seção 2.1 apresenta um histórico da computação, ou seja, as pesquisas desenvolvidas até chegar na computação pervasiva. A Seção 2.2 descreve o que é sensibilidade ao contexto e a sua importância na computação pervasiva. Por fim, a Seção 2.3 descreve mobilidade de código.

2.1 Histórico

A computação pervasiva define uma importante evolução que se iniciou na década de 70, da computação, quando os primeiros computadores para uso pessoal começaram a ser produzidos. Dois importantes passos nessa evolução são os sistemas distribuídos e a computação móvel (SATYANARAYANAN, 2001; SAHA; MUKHERJEE, 2003). As principais pesquisas na área de sistemas distribuídos foram realizadas entre a década de 70 até o início da década de 90. As pesquisas nessa área permitiram o acesso a informações e recursos remotos, a comunicação com tolerância a falhas, a alta disponibilidade de recursos e a segurança (criptografia na autenticação em redes de larga escala) (SATYANARAYANAN, 2001).

A linha de pequisa de computação móvel surgiu no início da década de 90, quando os primeiros *laptops* e redes sem fio começaram a surgir, fazendo com que novos desafios surgissem na tentativa de criar sistemas distribuídos com clientes móveis (SATYANARAYANAN, 2001). Conforme Satyanarayanan (2001), os conceitos básicos de sistemas distribuídos também se aplicam à computação móvel, porém, devido a quatro características únicas de sistemas móveis, novas técnicas necessitaram serem desenvolvidas. Essas características são: a variação imprevisível na qualidade da rede de comunicação, a baixa confiança e robustez de elementos móveis, as limitações impostas pelo tamanho e peso dos dispositivos, e a preocupação com o consumo de bateria. Para Saha e Mukherjee (2003), a computação móvel é considerada um subconjunto da computação pervasiva, porém, além das características citadas da computação móvel, os sistemas pervasivos necessitam ainda de suporte para interoperabilidade, escalabilidade, inteligência (*smartness*), e invisibilidade, a fim de garantir que o usuários tenham acesso às informações sempre que precisarem. A Tabela 2.1 resume as principais características da

computação distribuída, móvel e pervasiva.

Tabela 2.1 – Características conforme o tipo de computação. Adaptado de Satyanarayanan (2001)

	Distribuída	**Móvel**	**Pervasiva**
Comunicação Remota	**X**	**X**	**X**
Tolerância à falhas	**X**	**X**	**X**
Alta Disponibilidade	**X**	**X**	**X**
Acesso a informações remotas	**X**	**X**	**X**
Segurança	**X**	**X**	**X**
Redes móveis		**X**	**X**
Acesso a informações móveis		**X**	**X**
Suporte para aplicações adaptativas		**X**	**X**
Técnicas para economia de baterias		**X**	**X**
Sensibilidade à localização		**X**	**X**
Espaços inteligentes			**X**
Invisibilidade			**X**
Escalabilidade em todos ambientes			**X**
Mascarar variações nos ambientes			**X**

Com o advento da computação móvel e o rápido crescimento dessas redes, alguns problemas e desafios surgiram, como por exemplo, a escalabilidade (FUGGETTA; PICCO; VIGNA, 1998). Os nós em redes *wireless* podem mover-se e desconectar-se a qualquer instante, consequentemente a topologia da rede não pode ser definida estaticamente. Dessa forma, muitos dos princípios pesquisados em sistemas distribuídos não podem ser aplicados diretamente na computação móvel. Além disso, serviços de rede já são encontrados em vários segmentos da sociedade, o que torna necessário a sua customização, para que os usuários possam adaptá-los de acordo com as necessidades e preferências. Uma das abordagens que surgiram na tentativa de garantir a customização e flexibilidade das redes é a mobilidade de código (descrita na Seção 2.3).

2.2 Sensibilidade ao contexto

O uso da localização e de outras informações sobre o ambiente em sistemas pervasivos permite que a aplicação se adapte às necessidades do usuário, ou seja, a aplicação poderá ter diferentes comportamentos conforme o *contexto* na qual está inserida. Essa técnica é denominada *"sensibilidade ao contexto"* (WANT; PERING, 2005; MOSTÉFAOUI; PASQUIER-ROCHA; BRÉZILLON, 2004).

Segundo Lee (2009), a definição de contexto mais citada na bibliografia é a definição

proposta por Abowd *et al.*:

> Contexto é qualquer informação que possa ser utilizada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, um lugar ou um objeto que possam ser considerados relevantes para a interação entre um usuário e uma aplicação, incluindo o usuário e as suas próprias aplicações. (ABOWD et al., 1999, tradução nossa)

A importância do contexto na computação pervasiva é permitir adaptabilidade entre aplicações e dispositivos (MOSTÉFAOUI; PASQUIER-ROCHA; BRÉZILLON, 2004). A adaptabilidade pode ser um desafio levando em consideração a grande heterogeneidade de dispositivos nos ambientes pervasivos, assim como a comunicação pervasiva entre eles. Dispositivos heterogêneos possuem diferentes funcionalidades e não necessariamente possuem um mesmo protocolo padrão para comunicação. As adaptações ao contexto podem ser previstas estaticamente (durante o desenvolvimento da aplicação) ou dinamicamente (durante a execução do programa). Devido à grande heterogeneidade e à comunicação pervasiva entre os dispositivos móveis, as adaptações ao contexto devem ser feitas dinamicamente, garantindo maior flexibilidade e adaptabilidade das aplicações. A heterogeneidade pode ser de dados ou de comunicação (LEE; PARK; LEE, 2009). Um exemplo do uso de contexto em aplicações é citado por (HOFER et al., 2003): uma aplicação no estilo agenda, para controlar as tarefas de um usuário. A aplicação percebe o ambiente no qual está inserida (contexto). Caso o contexto atual seja uma sala de reuniões, um alerta vibratório é disparado ao invés de um alerta sonoro.

2.3 Mobilidade de código

Mobilidade de código é a capacidade de alterar dinamicamente a localização onde fragmentos de código são executados (FUGGETTA; PICCO; VIGNA, 1998). Existem duas formas de mobilidade: fraca e forte.

Na mobilidade fraca, o código é movido entre diferentes ambientes computacionais, e nenhuma informação sobre o estado atual da execução é transferida, somente algum dado opcional para a inicialização da execução no novo ambiente.Na mobilidade forte, além do código, o estado atual da execução também é transferido para o novo ambiente. A mobilidade forte pode ser implementada por dois mecanismos: migração ou clonagem remota. Na migração, o trecho de código que está sendo executado é suspenso, transferido para o ambiente de destino e a execução continua do ponto onde foi suspenso. Na clonagem remota, o trecho de código que está sendo executado é copiado para o ambiente de destino.

Conforme Augustin (2004), a computação pervasiva fornece o cenário mais geral de

mobilidade, pois todos os dispositivos envolvidos podem ser móveis. Além disso, ela permite a mobilidade lógica e física enquanto todos os dispositivos mantêm-se conectados.

2.4 Resumo do capítulo

Este capítulo resumiu os principais conceitos da computação pervasiva, descrevendo como surgiu, e suas principais características e desafios. A principal ideia da computação pervasiva é tornar natural o uso da computação, dando a ideia de invisibilidade. A sensibilidade ao contexto é um dos atuais desafios da computação pervasiva. As aplicações devem possuir comportamentos diferentes conforme o contexto no qual estão inseridas. Além disso, as adaptações ao contexto devem ocorrer sem intervenção do usuário. A mobilidade de código também é outro conceito importante para a computação pervasiva, pois as aplicações devem "seguir" o usuário independente do ambiente ou dispositivo que estejam utilizando. Um dos desafios da computação pervasiva é desenvolver um modelo padrão para modelar e utilizar contextos em aplicações.

3 SISTEMA DE TIPOS

A teoria dos tipos foi desenvolvida no início dos anos 1900 por Bertrand Russell, com o propósito inicial vinculado ao estudo da lógica e dos fundamentos da matemática (FARMER, 2008). A teoria dos tipos começou a ser utilizada em computação como uma ferramenta para garantir a consistência de dados e evitar a presença de erros em programas (WRIGHT, 2010).

Sistema de tipos (ou teoria dos tipos), na área da Ciência da Computação, é um método formal que ajuda a garantir que um sistema se comporta de acordo com a sua especificação. Sua principal função é prevenir a ocorrência de erros durante a execução de um programa (CARDELLI, 1996, 2004). Uma definição proposta por Pierce (2002) é: *"Um sistema de tipos é um método sintático tratável para provar a ausência de certos comportamentos em um programa através da classificação de frases de acordo com o tipo de valores que elas computam"*.

Em 1954, Fortran foi a primeira linguagem a distinguir números inteiros de números de ponto flutuante por restrições de *hardware* na época. A distinção entre os tipos numéricos era feita através da primeira letra utilizada para nomear a variável. Porém, a primeira linguagem a utilizar tipos explicitamente foi a linguagem Algol 60 (CARDELLI; WEGNER, 1985).

Um sistema de tipos não consegue antecipar que um programa está livre de comportamentos arbitrários e indesejados, só pode garantir que se ele está *"bem tipado"* ele está livre de alguns tipos de comportamentos. Por exemplo, um sistema de tipos pode verificar estaticamente que os argumentos utilizados em uma operação aritmética (como soma) são sempre números, mas não pode garantir que o segundo argumento em uma divisão é um número diferente de zero, ou que o acesso a uma posição de um vetor está dentro do limite do seu tamanho (PIERCE, 2002). Essas verificações também são de responsabilidade do sistema de tipos, porém devem ser analisadas dinamicamente, ou seja, conforme o programa é executado.

Em um programa, uma variável pode assumir diferentes valores (sempre dentro de uma faixa permitida) durante sua execução. Essa faixa permitida de valores é denominada de *tipo* da variável (CARDELLI, 2004). Um tipo, em linguagens de programação, é uma definição de um conjunto de valores (por exemplo, "todos os inteiros") e as operações permitidas para esses valores: multiplicação, adição, etc (WRIGHT, 2010). Linguagens que permitem a definição de tipos para as variáveis são chamadas de *linguagens tipadas*. Em contrapartida linguagens que não restringem a faixa de valores das variáveis são denominadas *linguagens não tipadas*.

Os sistemas de tipos também são categorizados em duas divisões: estáticos ou dinâmi-

cos. Em linguagens tipadas, a maioria dos erros e regras pré-definidas podem ser verificadas estaticamente, ou seja, em tempo de compilação. O algoritmo que faz essa verificação é chamado "*typechecker*" . Segundo Cardelli (1988), a ideia de adicionar sistema de tipos em linguagens de programação foi permitir a verificação estática dos programas. Porém, mesmo em linguagens de programação que possuam verificação estática de tipos, a verificação dinâmica também é necessária para garantir segurança. Um exemplo clássico é verificar se o acesso a uma posição de um vetor está dentro dos limites permitidos. Um programa que passa na verificação de tipos é denominado *bem tipado* (CARDELLI, 2004).

Algumas aplicações de sistemas de tipos em linguagens de programação são (PIERCE, 2002):

- **Detecção de erros:** A verificação estática de tipos permite detectar antecipadamente erros de programação, os quais podem ser corrigidos imediatamente, ao invés de serem descobertos posteriormente, às vezes somente quando o sistema já está terminado e em produção.

- **Abstração:** Os sistemas de tipos forçam uma programação disciplinada. Em sistemas de grande escala, os sistema de tipos formam a principal estrutura dos métodos utilizados em pacotes, que juntos representam os componentes do sistema. Uma interface pode ser vista como um tipo de módulo e é um exemplo de abstração.

- **Documentação:** Tipos são úteis para a leitura de programas. A declaração de tipos em cabeçalho de métodos e interfaces constitui uma forma de documentação, permitindo prever o comportamento do método.

- **Segurança:** Uma linguagem segura é uma linguagem que protege a suas abstrações. Esse item está relacionado à detecção de erros. Provar que um programa é livre de determinados tipos de comportamentos pode torná-lo mais seguro e confiável.

- **Eficiência:** Diferenciar valores inteiros de valores fracionados permite ao compilador usar diferentes representações e gerar apropriadas instruções de máquina para operações primitivas. Em linguagens seguras, a eficiência é obtida eliminando-se muitas das verificações dinâmicas que seriam necessárias para garantir a segurança da linguagem, provando estaticamente que elas sempre serão satisfeitas.

- **Provadores de teoremas:** Os sistemas de tipos podem ser utilizados para representar proposições lógicas e provas.

A forma utilizada para provar formalmente que um sistema de tipos garante que programas bem tipados estão livres de determinados comportamentos é através de provas matemáticas. Um teorema ou proposição vinculado ao sistema de tipos demostra que um programa bem tipado sempre terá o comportamento desejado. Essa propriedade deve ser provada matematicamente garantindo assim que o sistema de tipos é "*sound*" (CARDELLI, 2004) ou "*safe*" (PIERCE, 2002). Um sistema de tipos estático é "*sound*" se programas bem tipados não causam erros de execução (WRIGHT; FELLEISEN, 1994). Segundo Pierce (2002), existem duas propriedades importantes que ajudam a provar que um sistema de tipos é "*sound*":

- **Progressão (*progress*):** Essa propriedade diz que um termo bem tipado não "trava", ou seja, ou ele é um valor final, ou pode ser avaliado para gerar outro termo).

- **Preservação (*preservation*)**, também chamado de ***subject reduction***. Essa propriedade diz que se um termo bem tipado pode ser avaliado, então o resultado dessa avaliação também é um termo bem tipado.

3.1 Formalização de linguagens de programação

Segundo Cardelli (2004), o primeiro passo para formalizar uma linguagem de programação é definir a sua sintaxe - geralmente os seus tipos e seus termos. O próximo passo é definir as regras de escopo da linguagem. O escopo é formalizado definindo o conjunto de *variáveis livres* de um fragmento do programa. Por fim, as regras de tipos da linguagem podem ser definidas. Essas regras definem a relação *"possui o tipo"* na forma *M:A*, ou seja, o termo *M* e tipo *A*. As coleções de regras de tipos de uma linguagem formam o seu sistema de tipos.

Um conceito importante na formalização de tipos é o *ambiente estático de tipos* (geralmente definido pela letra grega Γ) que é utilizado para armazenar o tipo das variáveis livres durante o processamento de partes do programa. Um ambiente vazio é denotado através do símbolo $\varnothing$.

Um sistema de tipos é formalizado através de uma série de sentenças e regras. As sentenças possuem o formato $\Gamma \vdash \Im$, sendo que as variáveis livres de $\Im$ devem estar declaradas em Γ. A Figura 3.1 apresenta alguns exemplos de sentenças.

Sentenças

$\Gamma \vdash$ *M:A*	(*M* possui o tipo *A* em Γ)
$\varnothing \vdash$ *true:Bool*	(*true* possui o tipo *Bool*)
$\varnothing$, *x:Int* $\vdash$ *x*+1 : *Int*	(*x*+1 possui o tipo *Int*, desde que *x* tenha o tipo *Int*)
$\Gamma \vdash \diamond$	(ambiente Γ está *bem formado* ou construído)

Figura 3.1 – Exemplo de sentenças

Uma sentença por ser válida ou inválida. A validade da sentença formaliza a noção de programas *bem tipados*. A notação utilizada para formalizar as regras de tipos é modular, ou seja, regras para diferentes construções são escritas separadamente. As regras de tipos declaram a validade de certas sentenças com base em outras sentenças já conhecidas e que também são válidas. As regras possuem o seguinte formato (CARDELLI, 2004):

$$\frac{\text{(Nome da regra)}\ \ \text{(Anotações)}}{\Gamma \vdash \Im \ \ldots \ \Gamma_n \vdash \Im_n \quad \text{(Anotações)}} \qquad \Gamma \vdash \Im$$

Acima da linha horizontal estão as premissas da sentença e abaixo da linha uma única conclusão. O número de premissas pode ser zero. Nesse caso, a regra é um axioma e a linha horizontal não é obrigatória, somente existirá a conclusão. Quando todas as premissas de uma regra forem satisfeitas, a conclusão será válida.

A Figura 3.2 apresenta a gramática de uma linguagem, no qual será utilizada para exemplificar os conceitos básicos sobre sistemas de tipos. Nessa linguagem, são definidos dois conjuntos de expressões: M que é o conjunto de possíveis expressões que podem ser produzidas nessa linguagem, e v, o conjunto de valores terminais. Terminais (ou valores) não produzem novas expressões, ou seja, sinalizam o fim da expressão.

Sintaxe

$M ::=$	Termos
$true$	constante *true*
$false$	constante *false*
$0, 1, \ldots$	números inteiros positivos
$M + M$	soma de inteiros
$if\ M\ then\ M\ else\ M$	condicional *if*
$v ::=$	Valores
$true$	valor *true*
$false$	valor *false*
$0, 1, \ldots$	números inteiros positivos

Figura 3.2 – Sintaxe (ou gramática) de uma linguagem

A Figura 3.3 apresenta os tipos da linguagem, que podem ser $Bool$, para expressões booleanas e Int para os números inteiros.

Tipos

$T ::=$	Tipos
$Bool$	tipo para booleanos
Int	tipo para números inteiros positivos

Figura 3.3 – Tipos para a linguagem

A regra de tipos para booleanos, números inteiros, soma de números inteiros e a expressão condicional *if* podem ser formalizadas conforme as regras apresentadas na Figura 3.4.

Regras de tipos

(Tipo Bool) (Tipo Int)

$$\frac{}{\varnothing \vdash Bool} \qquad \frac{}{\varnothing \vdash Int}$$

(Val True) (Val False) (Val *n*) (*n* = 0, 1, . . .)

$$\frac{}{\varnothing \vdash true : Bool} \qquad \frac{}{\varnothing \vdash false : Bool} \qquad \frac{}{\varnothing \vdash n : Int}$$

(Val If)

$$\frac{\varnothing \vdash M : Bool \quad \varnothing \vdash N_1 : T \quad \varnothing \vdash N_2 : T}{\varnothing \vdash if\ M\ then\ N_1\ else\ N_2 : T}$$

(Val Soma)

$$\frac{\varnothing \vdash M : Int \quad \varnothing \vdash N : Int}{\varnothing \vdash M{+}N : Int}$$

Figura 3.4 – Regras de tipos para booleanos e inteiros

As regras “Tipo Bool” e “Tipo Int” informam que $Bool$ e Int são tipos válidos no sistema de tipos. As regras “Val True” e “Val False” associam o tipo *Bool* com as constantes *true* e *false*. Similarmente a regra “Val *n*” associa o tipo *Int* a qualquer número inteiro positivo. A regra “Val If” associa um tipo à expressão condicional *if* baseado no tipo de suas sub-expressões. Além disso, existem duas restrições nessa regra: a expressão a ser analisada, no caso *M* sempre terá que ser do tipo *Bool*, enquanto que N_1 e N_2 devem ser do mesmo tipo. Finalmente a regra “Val Soma” informa que se, o termo M é do tipo *Int* e se, o termo N também é do tipo *Int*, então a soma desses termos também resultará em um termo *Int*.

Por fim, é possível definir as regras de avaliação (semântica) para as expressões da linguagem. Nesse caso, a única expressão que pode ser avaliada é o *if*, conforme pode ser visualizado na Figura 3.5.

As duas primeiras regras são simples. Em “AvalIf-True” a regra informa que se a con-

Regras de avaliação

$if\ \ true\ \ then\ N_1\ else\ N_2 \rightarrow N_1$	(AvalIf-True)
$if\ false\ then\ N_1\ else\ N_2 \rightarrow N_2$	(AvalIf-False)
$\dfrac{M_1 \rightarrow M_1^{'}}{if\ M_1\ then\ N_2\ else\ N_3 \rightarrow if\ M_1^{'}\ then\ N_2\ else\ N_3}$	(AvalIf)

Figura 3.5 – Regras de avaliação para booleanos

dição do if for a constante *true* então o próximo passo é avaliar o termo N_1, ou seja, o que vier logo após o *then*. Em "AvalIf-False" o pensamento é o oposto: se a condição do if for a constante *false* então o próximo passo é avaliar o termo que vier logo após o *else*, no caso N_2. Por fim, em "AvalIf" a regra informa que se a expressão M_1 a ser avaliada não for um valor, então ela precisa ser avaliada gerando um valor ou termo $M_1^{'}$ e assim a avaliação prossegue.

Para verificar se uma sentença é bem tipada, utiliza-se o conceito de "árvore de derivação". Uma derivação deve ser lida de baixo para cima, o seja, utilizando o conceito de árvore. Cada sentença da derivação é obtida por alguma regra do sistema de tipos. Uma sentença será válida se sempre existir alguma regra que permita expandir a derivação, até chegar em um nó folha. Por exemplo, como verificar se a sentença $if\ true\ then\ 0\ else\ 1 : Int$ é válida. A prova que a sentença é válida pode ser visualiza na seguinte árvore de derivação:

$$\dfrac{\dfrac{}{\varnothing \vdash true : Bool}\ \text{(Val True)} \quad \dfrac{}{\varnothing \vdash 0 : Int}\ \text{(Val } n\text{)} \quad \dfrac{}{\varnothing \vdash 1 : Int}\ \text{(Val } n\text{)}}{\varnothing \vdash if\ true\ then\ 0\ else\ 1 : Int}\ \text{(Val If)}$$

Um tipo básico em linguagens de programação é o tipo *Unit* (CARDELLI, 2004), utilizado quando o tipo de algum argumento ou o resultado de uma função não precise ser informado. O tipo *Unit* é como o *void*, utilizado em linguagens de programação como *C* e *Java* (PIERCE, 2002). A Figura 3.6 apresenta as regras de tipo para o tipo *Unit*.

Tipos Unit

(Tipo Unit)	(Val Unit)
$\dfrac{}{\varnothing \vdash Unit}$	$\dfrac{}{\varnothing \vdash unit : Unit}$

Figura 3.6 – Regras de tipo para Unit

A regra "Tipo Unit" somente informa que *Unit* é um tipo bem formado no ambiente Γ, e "Val Unit" informa que *unit* é um valor bem formado do tipo *Unit*.

3.2 Cálculo *lambda*

O Cálculo *lambda* ou Cálculo-λ é um modelo de computação criado por Alonzo Church (1940), no qual toda a computação é reduzida a operações básicas como funções e aplicações. O Cálculo-λ é utilizado para especificar características de linguagens de programação e no estudo de sistemas de tipos (PIERCE, 2002). A sintaxe completa do Cálculo-λ é composta por apenas três termos (categoria sintática M), conforme visualizado na Figura 3.7.

Sintaxe Cálculo-λ tipado

$T ::=$	Tipos
$Bool$	tipo booleano
Int	tipo inteiro
$T \rightarrow T$	tipo para as funções
$M ::=$	Termos
x	variável
$\lambda x{:}T.M$	abstração
$M\ M$	aplicação

Figura 3.7 – Sintaxe do Cálculo Lambda tipado

Em uma linguagem de programação, uma função *fatorial(x)* que calcule o fatorial de um número *x* pode ser escrita como a abstração $\lambda x{:}T.Y$, sendo Y o corpo da função, ou seja, a lógica utilizada para calcular o fatorial, e *x* o valor do fatorial a ser calculado, ou seja, o argumento da função. Por exemplo, Y poderia ser escrito da seguinte forma:

$$Y \ = \ \textbf{if}\ (x == 0)\ \textbf{then}\ 1\ \textbf{else}\ (x * Y(x-1))$$

Note que *x* é utilizada dentro do corpo de Y. Uma *aplicação* dessa função, por exemplo *fatorial(3)* por ser escrita como:

$$\overbrace{\underbrace{(\lambda x : Int.Y)}_{M}\ \underbrace{(3)}_{M}}^{\text{aplicação}}$$

Nesse caso, *x*, que está ligada na abstração λ (ou seja, é o parâmetro da função), será substituída pelo valor 3. Ou de uma forma mais formal: $Y[3/x]$ (Substitua x por 3 no corpo de Y).

Em se tratando de escopo de variáveis, uma ocorrência da variável x é denominada *ligada* quando ela aparece no corpo M de uma abstração $\lambda x.M$, ou seja, está ligada pela abstração. A ocorrência de x é *livre* quando ela aparece em uma posição em que não está ligada

por uma abstração (PIERCE, 2002). Por exemplo, em $\lambda y.xy$, x é uma variável livre, enquanto y é ligada pela abstração. Um termo sem variáveis livres é denominado *fechado* ou *combinador*. O combinador mais simples é a *função identidade* que é uma função que apenas retorna o seu argumento, independente do termo passado por parâmetro:

$$id = \lambda x.x \qquad \text{(função identidade)}$$

As regras de tipos para o Cálculo-λ tipado podem ser visualizadas nas Figura 3.8 (PIERCE, 2002). Como as abstrações podem possuir várias variáveis, é necessário utilizar o ambiente Γ, que guardará a lista de variáveis livres dos termos, com seus respectivos tipos.

Tipos Cálculo-λ

(Tipo Var)	(Tipo Abs)	(Tipo Aplic)
$\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$	$\dfrac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : T_1 \to T_2}$	$\dfrac{\Gamma \vdash M_1 : T_{11} \to T_{12} \qquad \Gamma \vdash M_2 : T_{11}}{\Gamma \vdash M_1\ M_2 : T_{12}}$

Figura 3.8 – Regras de tipos para o Cálculo Lambda tipado

A regra "Tipo Var" significa que o tipo assumido por x em Γ é T. A regra "Tipo Abs" informa que uma vez conhecido o tipo do argumento da função (no caso T_1), e o tipo do corpo da função (T_2), então é possível concluir que o resultado da função será o mesmo do corpo de M, ou seja T_2, onde as ocorrências de x em M sempre terão o tipo T_1. Ou seja, é uma função que transforma entradas dos tipo T_1 em saídas do tipo T_2. Por fim, a regra "Tipo Aplic" informa que se M_1 é uma função que transforma entradas do tipo T_{11} em saídas do tipo T_{12} e M_2 avalia para T_{11} então o tipo do resultado da aplicação de M_1 em M_2 será T_{12}.

3.3 Correspondência *Curry-Howard*

As regras de sistema de tipos possuem duas categorias: uma regra de introdução (com regras descrevendo como elementos do tipo podem ser criados) e uma regra de eliminação (descrevendo como elementos do tipo podem ser utilizados). Por exemplo, no Cálculo-λ tipado, descrito na Seção 3.2, a regra de tipo "Tipo Abs" descreve como elementos do tipo podem ser criados e "Tipo Aplic" descreve como elementos do tipo podem ser utilizados.

A terminologia de introdução e de eliminação provém de uma conexão entre a teoria dos tipos e a lógica, chamada de Correspondência *Curry-Howard* ou isomorfismo *Curry-Howard* (PIERCE, 2002). Na lógica construtiva, a prova de uma proposição P consiste em uma evidência concreta para P. Curry e Howard verificaram que essa "evidência" tem uma

grande relação com a computação. Por exemplo, $P \supset Q$ (P implica em Q ou $P \rightarrow Q$) pode ser visto como um procedimento que dada uma prova de P, constrói uma prova de Q. A correspondência completa pode ser visualizada na Tabela 3.1 (PIERCE, 2002):

Tabela 3.1 – Correspondência *Curry-Howard*

Lógica	**Linguagens de Programação**
proposições	tipos
proposição $P \supset Q$	tipo $P \rightarrow Q$ (função)
proposição $P \wedge Q$	tipo de produto $P \times Q$
prova de uma proposição P	termo t do tipo P (ou seja, $t : P$)
proposição P é provável	tipo P é habitado por algum termo

3.4 Tipos estruturados

Além dos tipos básicos, linguagens de programação oferecem tipos estruturados, como tipos de produto, tipos de união, tipos de registros e tipos variáveis, que serão descritos nas próximas subseções.

3.4.1 Tipos de Produto

Tipos de produto (CARDELLI, 2004), também chamados de pares (PIERCE, 2002), são descritos na forma $T_1 \times T_2$. Um tipo de produto é um par no qual o primeiro componente se refere ao tipo T_1 e o segundo ao tipo T_2. Para acessar os valores que estão dentro dos pares, pode-se utilizar a estrutura *with*, que decompõe um par *M* e vincula seus dois componentes em variáveis x_1 e x_2 no escopo *N*. A Figura 3.9, formaliza as regras para tipos de produto (CARDELLI, 2004).

Seguindo a ideia da correspondência *Curry-Howard* descrita na Seção 3.3, "Val Par" é uma regra de introdução de tipos enquanto que "Val First", "Val Second" e "Val With" são regras de eliminação, ou seja, demonstram como os elementos do tipo podem ser utilizados.

Tipos de produto podem ser generalizados em tuplas de tamanho *n* $T_1 \times \ldots \times T_n$, por exemplo, $\langle 1, 2, true \rangle$, que seria uma tupla de tipo $\langle Int, Int, Bool \rangle$. Para utilizar essa notação de tupla, é necessário apenas ajustar a regra de tipo "Val With", para torná-la genérica, ou seja, para aceitar qualquer tamanho de tupla.

Tipos de produto

(Tipo de Produto)
$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 \times T_2}$$

(Val Par)
$$\frac{\Gamma \vdash M_1 : T_1 \qquad M_2 : T_2}{\Gamma \vdash \langle M_1, M_2 \rangle : T_1 \times T_2}$$

(Val First)
$$\frac{\Gamma \vdash M : T_1 \times T_2}{\Gamma \vdash first\ M : T_1}$$

(Val Second)
$$\frac{\Gamma \vdash M : T_1 \times T_2}{\Gamma \vdash second\ M : T_2}$$

(Val With)
$$\frac{\Gamma \vdash M : T_1 \times T_2 \qquad \Gamma, x_1{:}T_1, x_2{:}T_2 \vdash N : S}{\Gamma \vdash (with(x_1{:}T_1, x_2{:}T_2) := M\ do\ N) : S}$$

Figura 3.9 – Regras de tipos para tipos de produto

3.4.2 Tipos de União

Tipos de união (CARDELLI, 2004) ou tipos de soma (PIERCE, 2002), são descritos na forma T_1+T_2. Um tipo de união pode ser um elemento do tipo T_1 rotulado por um símbolo *left* (criado por *inLeft*) ou um elemento do tipo T_2 rotulado por um símbolo *right* (criado por *inRight*). Tipos de união são úteis para se trabalhar com coleções com tipos heterogêneos. Nesse caso, a variável vai ser do tipo T_1, caso esteja com o rótulo *inLeft*, ou do tipo T_2, caso esteja com o rótulo *inRight*. A Figura 3.10, formaliza as regras para tipos de união (CARDELLI, 2004).

Tipos de união

(Tipo de União)
$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 + T_2}$$

(Val inLeft)
$$\frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash inLeft_{T_1}\ M_1 : T_1 + T_2}$$

(Val inRight)
$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash M_2 : T_2}{\Gamma \vdash inRight_{T_2}\ M_2 : T_1 + T_2}$$

(Val Case)
$$\frac{\Gamma \vdash M : T_1 + T_2 \qquad \Gamma, x_1{:}T_1 \vdash N_1 : S \qquad \Gamma, x_2{:}T_2 \vdash N_2 : S}{\Gamma \vdash (case\ M\ of\ x_1{:}T_1\ then\ N_1 \mid x_2{:}T_2\ then\ N_2) : S}$$

Figura 3.10 – Regras de tipos para tipos de união

A regra “Val Case” executa um dos ramos, separados por uma barra vertical |, dependendo do rótulo de *M*. Dessa forma, *M* receberá o valor de x_1 ou de x_2 no escopo de N_1 ou de N_2.

Conforme Cardelli (2004), tipos de produtos e de união podem ser mesclados para produzirem tuplas e uniões múltiplas, porém são complexos de se utilizar e raramente vistos em linguagens de programação. Para contornar essa complexidade existem os tipos de registro e os

tipos variáveis.

3.4.3 Tipos de Registro

Tipos de registro (*record types*) são coleções de tipos rotulados com um nome, e possuem operações que possibilitam extrair um componente através de seu nome. A Figura 3.11, formaliza as regras para tipos de registro (CARDELLI, 2004).

Tipos de Registro

(Tipo de Registro) (l_i distinto)

$$\frac{\Gamma \vdash T_1 \quad \ldots \quad \Gamma \vdash T_n}{\Gamma \vdash Record(l_1{:}T_1,\ \ldots, l_n{:}T_n)}$$

(Val Seleciona Registro)

$$\frac{\Gamma \vdash M : Record(l_1{:}T_1,\ \ldots, l_n{:}T_n) \qquad j \in 1..n}{\Gamma \vdash M.l_j : T_j}$$

(Val Registro) (l_i distinto)

$$\frac{\Gamma \vdash M_1 : T_1 \quad \ldots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash record(l_1{=}M_1,\ \ldots, l_n{=}M_n) : Record(l_1{:}T_1,\ \ldots, l_n{:}T_n)}$$

(Val Registro With)

$$\frac{\Gamma \vdash M : Record(l_1{:}T_1,\ \ldots, l_n{:}T_n) \qquad \Gamma, x_1{:}T_1,\ \ldots, x_n{:}T_n \vdash N : S}{\Gamma \vdash (with\ (l_1{=}x_1{:}T_1,\ \ldots, l_n{=}x_n{:}T_n) := M\ do\ N) : S}$$

Figura 3.11 – Regras de tipos para tipos de registro

A regra "Val Registro With" extrai os componentes do registro através dos rótulos $l_1,\ \ldots, l_n$, atribuindo os valores para as variáveis $x_1,\ \ldots, x_n$ no escopo N. Tipos de produto $T_1 \times T_2$ podem ser definidos na forma de registros: $Record(first{:}T_1, second{:}T_2)$.

3.4.4 Tipos Variáveis

Tipos variáveis (*variant types*), também chamados de uniões disjuntas dos tipos (*disjoint unions of types*), são uma generalização para os tipos de união, assim como tipos de registro são para tipos de produto (PIERCE, 2002). Os componentes são identificados pela ordem em que estão dentro do *Variant*. A Figura 3.12, formaliza as regras para tipos variáveis (CARDELLI, 2004).

Tipos de união $T_1{+}T_2$ podem ser escritos como $Variant(left{:}T_1, right{:}T_2)$. Tipos enumerados como $\{$*vermelho, verde, azul*$\}$ podem ser definidos como:

$$Variant(vermelho : Unit, verde : Unit, azul : Unit)$$

Tipos Variáveis

(Tipo Variável) (l_i distinto)

$$\frac{\Gamma \vdash T_1 \quad \ldots \quad \Gamma \vdash T_n}{\Gamma \vdash Variant(l_1{:}T_1,\ \ldots, l_n{:}T_n)}$$

(Val Variável) (l_i distinto)

$$\frac{\Gamma \vdash T_1 \quad \ldots \quad \Gamma \vdash T_n \qquad \Gamma \vdash M_j : T_j \qquad j \in 1..n}{\Gamma \vdash variant_{(l_1:T_1,\ \ldots, l_n:T_n)}(l_j{=}M_j) : Variant(l_1{:}T_1,\ \ldots, l_n{:}T_n)}$$

(Val Variável Case)

$$\frac{\Gamma \vdash M : Variant(l_1{:}T_1,\ \ldots, l_n{:}T_n) \qquad \Gamma, x_1{:}T_1 \vdash N_1 : S \quad \ldots \quad \Gamma, x_n{:}T_n \vdash N_n : S}{\Gamma \vdash (case\ M\ of\ l_1{=}x_1{:}T_1\ then\ N_1 \mid \ldots \mid l_n{=}x_n{:}T_n\ then\ N_n) : S}$$

Figura 3.12 – Regras de tipos para tipos de variáveis

3.5 Quantificadores

3.5.1 Quantificador Universal

Os recursos de sistemas de tipos apresentados nas seções anteriores modelam tipos e funções monomórficas. Função polimórficas são funções que podem ser utilizadas com diferentes tipos (SEBESTA, 2009; WATT, 2004). A função identidade $\lambda x.x$ apresentada na seção 3.2 é um exemplo de função polimórfica, e não é dependente de tipos, ou seja, possui o mesmo comportamento para qualquer tipo de dado: somente retorna o argumento recebido. Porém, com a introdução do Cálculo-λ tipado, as variáveis ligadas devem possuir um tipo. Nesse caso, a função identidade tipada $(\lambda x{:}T).x$ deve ser escrita diversas vezes para representar todos os casos possíveis de tipos (Figura 3.13).

Função identidade tipada

$(\lambda x{:}Int).x$	(função identidade para inteiros)
$(\lambda x{:}Bool).x$	(função identidade para booleanos)
$(\lambda x{:}Int \to Int).x$	(função identidade que recebe uma função $Int \to Int$)
$(\lambda x{:}Int \to Bool).x$	(função identidade que recebe uma função $Int \to Bool$)
$\vdots$	

Figura 3.13 – Função identidade sem uso de quantificadores

Esta redundância fere o princípio da abstração da engenharia de software, que define que cada trecho de uma funcionalidade em um programa deve ser programada em apenas um lugar no código fonte (PIERCE, 2002).

Para expressar que a função identidade é a mesma para todos os tipos da linguagem é utilizado o quantificador *universal*, representado pelo símbolo $\forall$ (CARDELLI; WEGNER, 1985). O tipo da função identidade tipada $(\lambda x{:}T).x$ deve ser convertido para um tipo parametrizável, abstraindo T, sendo então escrita como:

$$id = (\lambda T.\lambda x{:}T).x$$

Assim, antes de utilizar a função é necessário informar o tipo que será substituído em T (CARDELLI, 2004), produzindo de volta $(\lambda x{:}T).x$ [2]:

$$id[Int] = (\lambda x{:}Int).x$$

por exemplo,

$$id[Int](5) = 5$$
$$id[Bool](true) = true$$

escrevendo de uma forma genérica:

$$id[T] = (\lambda x{:}T).x$$

Nas aplicações, além de passar os valores que serão substituídos nas abstrações, caso a função seja polimórfica, também deverão ser passados os tipos a serem substituídos nos argumentos das funções. Funções que necessitam de um tipo antes de poderem ser utilizadas são chamadas de funções genéricas (CARDELLI; WEGNER, 1985). Com esse novo conceito, novos termos devem ser adicionados no Cálculo-λ tipado, ficando a sintaxe completa conforme Figura 3.14.

Com base na nova sintaxe o tipo de um termo na forma $\lambda X.M$ é escrito como $\forall X.T$, significando que *para todo* X, o corpo da função M possuirá o tipo T (sendo que M e T poderão conter ocorrências de X) (CARDELLI, 2004). Por exemplo, o tipo da função identidade é dado por:

$$id : \forall T.T \rightarrow T$$

[2] A convenção utilizada por Cardelli e Wegner (1985) é que os parâmetros de tipos são passados dentro de colchetes [], enquanto os valores de parâmetros tipados são passados dentro de parênteses ().

Sintaxe do Cálculo-λ tipado com quantificador $\forall$

$T ::=$	Tipos
X	variável de tipo
$Bool$	tipo booleano
Int	tipo inteiro
$T \rightarrow T$	tipo para as funções
$\forall X.T$	tipo com quantificador universal
$M ::=$	Termos
x	variável
$\lambda x{:}T.M$	abstração
$M\ M$	aplicação
$\lambda X.M$	abstração polimórfica
$M[T]$	instanciação de tipos

Figura 3.14 – Sintaxe do Cálculo Lambda com quantificadores

Um exemplo de uma abstração que recebe uma função genérica por parâmetro (nesse exemplo, a função identidade), e cujo corpo da função é um "Tipo de produto", conforme descrito na Seção 3.4.1 pode ser escrito da seguinte forma:

$$funcao = \lambda(f : \forall T.T \rightarrow T).(f[Int], f[Bool])$$

Utilizando os operadores (Val First) e (Val Second) apresentados na Seção 3.4.1, pode-se obter duas instâncias diferentes da função identidade, uma para Int e outra para $Bool$:

$$\begin{aligned} first(funcao(id)) &: Int \rightarrow Int \\ second(funcao(id)) &: Bool \rightarrow Bool \end{aligned}$$

Conforme Cardelli e Wegner (1985), parâmetros do tipo "função genérica", ou seja, com quantificador universal, são úteis quando eles devem ser utilizados com diferentes tipos no corpo de uma mesma função. Por exemplo, uma função *tamanho* que retorna o tamanho de uma lista (ou vetor) qualquer.

$$tamanho : \forall T.[T] \rightarrow Int$$

Dessa forma, a função *tamanho* poderá receber por parâmetro listas de qualquer tipo, sempre retornando o seu tamanho.

3.5.2 Quantificador Existencial

Além dos tipos universais que possuem a forma $\forall X.T$, existem os tipos existenciais que possuem a forma $\exists X.T$, significando que *existe* um tipo X, tal que $\exists X.T$ possui o tipo T (CARDELLI; WEGNER, 1985). Por exemplo, considere o par $(3, 4)$:

$$(3, 4) : \exists T.T \times T$$
$$(3, 4) : \exists T.T$$

onde $T = Int$ no primeiro caso e $T = Int \times Int$ no segundo caso.

Conforme Cardelli e Wegner (1985), todo valor possuirá o tipo $\exists T.T$, pois para cada valor sempre existirá um tipo que irá satisfazer a regra (um valor sempre terá um tipo), ou seja, $\exists T.T$ descreve o conjunto de *todos* os tipos. Dessa forma nem todos os tipos existenciais são úteis. Se uma aplicação receber um objeto do tipo $\exists T.T$ não será possível manipulá-lo, pois não terá informações suficientes sobre o mesmo, pois conforme já citado vários tipos podem satisfazer $\exists T.T$. Por exemplo, se aplicação receber um objeto $\exists T.T \times T$, a única informação que se saberá é que é um Tipo de Produto (ver Seção 3.4.1) e pode-se utilizar os operadores (Val First) e (Val Second) para retornar o primeiro e segundo objeto do par respectivamente, porém mais nada poderá ser aplicado, pois não se têm mais informações do objeto (se é um Int e pode-se aplicar um operador *soma*, se é uma lista e pode-se aplicar o operador *tamanho*, etc.)

Tipos existenciais podem ser úteis se forem bem estruturados, por exemplo, $\exists T.T \times (T \rightarrow Int)$ é um tipo que possui informações suficientes para permitir a manipulação dos valores. Nesse tipo, o operador (Val Second) sempre retornará um Int.

Tipos existenciais são utilizados na construção de *tipos abstratos*, que abstraem algumas informações sobre o objeto que eles representam, porém possuem informações suficientes que permitem a sua manipulação. Por exemplo, o tipo $\exists T.T \times (T \rightarrow Int)$ pode ser visto como um *tipo abstrato* empacotado com um conjunto de operações. A variável T é o tipo abstrato que abstrai uma representação (de um Int, de um $Bool$, de um $[Int]$, etc.) A operação permitida sobre o tipo é representada por $T \times (T \rightarrow Int)$ (uma constante T e o operador $T \rightarrow Int$).

Para facilitar a representação, pode-se utilizar Tipos de Registro (ver Seção 3.4.3) para representar os tipos abstratos, podendo assim nomear a constante e o operador (CARDELLI;

WEGNER, 1985):

$$x : \exists T.Record(cons : T, operacao : T \rightarrow Int)$$

$$x.operacao(x.cons)$$

Como o tipo é existencial, não se possui informação que tipo será utilizado em T[3], não permitindo assim que x seja utilizado de forma livre e não trazendo muitas vantagens ao programador.

Um objeto do tipo $Record(3, succ)$ pode ser convertido para um objeto abstrato possuindo o tipo $\exists T.T \times (T \rightarrow Int)$. A conversão é feita *empacotando* esse objeto através da operação *pack*, assim, algumas de suas estruturas ficarão ocultas, conforme Figura 3.15.

Criando objetos existenciais

$interface = \exists T.Record(cons : T, operacao : T \rightarrow Int)$
$modulo : interface$
$\textbf{\textit{pack}}_{interface}\ T = Int\ \ \textbf{\textit{with}}\ \ record(3, succ)$

Figura 3.15 – Exemplo de criação de objetos existenciais

Na primeira linha, é criada uma $interface$ que declara que existe um tipo T (sem revelar a identidade interna) que possui uma constante e uma operação que retornará o sucessor dessa constante. Na segunda linha, uma variável desse tipo é criada. Na terceira linha, uma implementação específica da interface é criada para o tipo Int e o objeto $record(3, succ)$ é empacotado (com o operador *pack*).

Para utilizar o objeto, deve-se primeiramente abri-lo, através da operação *open*, acessando o registro através de x, conforme Figura 3.16.

Utilizando objetos existenciais

$\textbf{\textit{open}}_{Unit}\ modulo$
as $T,\ x : Record(cons : T, operacao : T \rightarrow Int)$
in $x.operacao(x.cons)$

Figura 3.16 – Exemplo de uso de objetos existenciais

Nesse exemplo, o tipo Int foi empacotado junto com a estrutura, dessa forma, é possível obter todas as informações necessárias para a manipulação do objeto quando ele for aberto. O tipo também poderia ser passado no momento da abertura do pacote, no momento de utilização

[3] Ao contrário dos tipos universais no qual obriga que a função funcione perfeitamente para qualquer tipo da linguagem, tipos existenciais não possuem essa restrição. A única informação que pode ser obtida é que *existe* pelo menos *um* tipo que irá satisfazer a função, mas não é possível saber previamente.

do operador $open$. As regras para empacotamento e abertura dos pacotes são formalizadas na Figura 3.17 (CARDELLI, 2004).

Tipos Existenciais

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \exists X.A}\ (\text{Tipo Existencial}) \qquad \frac{\Gamma \vdash [B/X]M : [B/X]A}{\Gamma \vdash (pack_{\exists\ X.A} X{=}B\ with\ M) : \exists X.A}\ (\text{Val Pack})$$

$$\frac{\Gamma \vdash M : \exists X.A \qquad \Gamma, X, x{:}A \vdash N : B \qquad \Gamma \vdash B}{\Gamma \vdash (open_B\ M\ as\ X, x{:}A\ in\ N) : B}\ (\text{Val Open})$$

Figura 3.17 – Regras de tipos para tipos existenciais

Resumindo, o quantificador universal produz *tipos genéricos* ou *funções genéricas*, enquanto o quantificador existencial produz *tipos de dados abstratos*. Os dois quantificadores podem ser utilizados em conjunto, produzindo *abstrações de dados parametrizáveis* (*parametric data abstractions*) (CARDELLI; WEGNER, 1985).

3.6 Tipos dependentes

Os tipos dependentes surgiram através da teoria intuicionista dos tipos (também conhecida como teoria dos tipos construtiva ou simplesmente de teoria Martin-Löf), proposta por Martin-Löf (1972). Tipos dependentes são tipos que dependem dos valores de uma ou mais variáveis que eles possuem (THOMPSON, 1991), ou seja, são tipos expressos em termos de dados (ALTENKIRCH; MCBRIDE; MCKINNA, 2005).

Ulf Norell, em sua tese, explica de uma maneira mais informal e mais clara o que são tipos dependentes:

> Na teoria não-dependente, tipos e termos vivem em mundos separados e eles somente se encontram para decidir quais termos possuem quais tipos. Na teoria dependente, por outro lado, tipos podem conversar com os termos [...] (NORELL, 2007, p. 14, tradução nossa)

Funções básicas, escritas na forma $A \to B$, podem ser convertidas para funções dependentes de tipos (*dependent function types*) escritas na forma $(x : A) \to B$, ou seja, o resultado de B depende do valor do argumento (SETZER, 2007). Tipos de funções dependentes também são chamadas de tipos-Π (NORELL, 2007).

Já os tipos de produto $A \times B$ podem ser generalizados em tipos de pares dependentes (*type of dependent pairs*), escritos na forma $(x : A) \times B$, onde o tipo do segundo componente

pode depender do valor do primeiro componente. Tipos de pares dependentes são chamados tipos-Σ e podem ser escritos como $\Sigma(A, B)$ para o par dependente (a, b), e π_1 sendo associado à projeção do primeiro componente do par π_2 ao segundo componente (SETZER, 2007; WAHLSTEDT, 2007). A Tabela 3.2 resume as principais diferenças entre tipos dependentes e não-dependentes.

Tabela 3.2 – Resumo de tipos dependentes e não-dependentes

Teoria dos tipos	**Teoria dependente**	**Sigla**
Função $A \to B$	Função dependente $(x : A) \to B$	$\Pi(A, B)$
Tipo de produto $A \times B$	Par dependente $(x : A) \times B$	$\Sigma(A, B)$

Conforme descrito na Seção 3.3, em sistema de tipos sempre existe uma regra de introdução (como elementos do tipo são criados) e uma de eliminação (como elementos do tipo são utilizados). Dessa forma, os tipos-Π e tipos-Σ podem ser formalizados conforme Figura 3.18 (DAPOIGNY; BARLATIER, 2009).

Regras de tipos dependentes

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ (\Pi\text{ - Intro}) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \langle M, N \rangle : \Sigma x : A.B}\ (\Sigma\text{ - Intro})$$

$$\frac{\Gamma \vdash \sigma : \Sigma x : A.B}{\Gamma \vdash \pi_1(\sigma) : A}\ (\pi_1\text{ - Elim}) \qquad \frac{\Gamma \vdash \sigma : \Sigma x : A.B}{\Gamma \vdash \pi_2(\sigma) : B[\pi_1(\sigma)/x]}\ (\pi_2\text{ - Elim})$$

Figura 3.18 – Regras de tipos para tipos dependentes

3.6.1 Tipos de Registros Dependentes

Uma extensão da teoria de Martin-Löf são os tipos de registros dependentes (*dependent record types*). Tipos de registros dependentes (TRD) baseiam-se nos tipos de registro, descritos na Seção 3.4.3, ou seja, são tuplas com rótulos. TRD é uma sequência de campos no qual l_i corresponde a um tipo T_i, e cada campo sucessivo pode depender dos valores dos campos anteriores (DAPOIGNY; BARLATIER, 2007):

$$\langle l_1 : T_1, l_2 : T_2(l_1) \ldots, l_n : T_n(l_1 \ldots l_{n-1}) \rangle$$

Por exemplo, $\langle n : Nat,\ v : Vetor(n) \rangle$ é um TRD e $\langle n = 2,\ v = [5, 6] \rangle$ é uma instância deste TRD que respeita o conceito de dependência, pois v possui o tipo $Vetor(2)$ (LUO, 2009).

3.7 Resumo do capítulo

Neste capítulo foram abordados os principais conceitos de sistema de tipos, tais como, histórico, vantagens, formalização e principais tipos utilizados em linguagens de programação. Sistemas de tipos, quando bem definidos, auxiliam na tarefa de detectar erros em programas, principalmente quando a verificação é realizada estaticamente. Nesse caso, o programador pode corrigir o erro antes que ele ocorra em tempo de execução. A maioria das linguagens de programação atuais utiliza sistema de tipos bem definidos na sua estrutura, tornando a linguagem mais segura para o programador.

4 CÁLCULO DE AMBIENTES

Com o crescimento da *Web* e os dispositivos móveis estando cada vez mais presentes no cotidiano, uma nova característica da computação emerge: a mobilidade de código. A grande dificuldade de mover códigos através da *Web* está no tratamento dos domínios administrativos (CARDELLI; GORDON, 1998, 2000). Não basta somente conhecer o endereço IP do dispositivo de destino. *Firewalls* e políticas de segurança controlam os domínios, restringindo o que pode passar através deles. Dessa forma, a mobilidade envolve além da autorização para executar aplicações ou para acessar informações: envolve autorização para entrar ou para sair de um certo domínio.

Com essa motivação, Cardelli e Gordon (1998) propõem um novo paradigma e modelo teórico para descrever mobilidade, chamado Cálculo de Ambientes (CA), que será descrito na próxima seção. Como a mobilidade de código é um requisito fundamental da computação pervasiva, descrita no Capítulo 2, o CA é um interessante modelo que pode ser utilizado para descrever o comportamento de aplicações pervasivas, principalmente quando adicionado suporte à sensibilidade ao contexto, conforme será descrito na Seção 4.4.

4.1 Cálculo de Ambientes Sem Tipos

Um ambiente é um lugar onde processos podem trocar mensagens e onde outros ambientes podem entrar e sair. A ideia é modelar as entidades do mundo real como ambientes. Analogamente à Programação Orientação a Objetos (POO) (POKKUNURI, 1989), onde a unidade básica são os objetos, no CA a unidade básica são os ambientes. Dessa forma, a abordagem utilizada por Cardelli e Gordon (1998) é estruturar hierarquicamente ambientes computacionais, confinando agentes a esses ambientes e movendo os ambientes através do controle desses agentes. Uma das vantagens, segundo os autores, é que essa abordagem permite mover também os subambientes, que podem estar contidos dentro dos ambientes a serem movidos, ao invés de mover somente objetos individualmente. O CA foi inspirado no Cálculo-π (MILNER; PARROW; WALKER, 1992; MILNER, 1999; SANGIORGI; WALKER, 2001). O Cálculo-π é um modelo teórico de computação (assim como o Cálculo-λ descrito na Seção 3.2) utilizado para descrever o comportamento de sistemas concorrentes, ou seja, sistemas distribuídos. A sintaxe do Cálculo-π permite representar processos, executando em paralelo, comunicação de proces-

sos através de canais, criação de novos canais e replicação de processos (WING, 2002). Um processo é uma abstração para uma tarefa independente qualquer. Um canal é uma abstração de um elo de comunicação entre dois processos. Dessa forma, os processos interagem enviando e recebendo mensagens através dos canais de comunicação. O CA estende o Cálculo-π adicionando novas primitivas para permitir mobilidade. A Figura 4.1 apresenta a sintaxe do CA.

Sintaxe

$P, Q ::=$	processos
$(\nu n)P$	restrição
$\mathbf{0}$	inatividade
$!P$	replicação
$P \mid Q$	composição
$M[P]$	ambiente
$M.P$	ação
$(x).P$	entrada
$\langle M \rangle$	saída assíncrona
$M ::=$	capacidades
x	variável
n	nomes
$in\ M$	entrar em ***M***
$out\ M$	sair de ***M***
$open\ M$	abrir ***M***
ε	nulo
$M.M'$	caminho

Figura 4.1 – Sintaxe do Cálculo de Ambientes

Conforme descrito na sintaxe, um ambiente é escrito na forma $M[P]$, onde M é o nome do ambiente [4] e P é o processo executando dentro desse ambiente. O processo P pode ser uma composição de vários outros processos ou ambientes. O operador binário $P|Q$ representa dois processos rodando em paralelo. Esse operador é comutativo e associativo. O operador de replicação $!P$ indica que o processo (ou serviço) P fará uma cópia de si mesmo a cada vez que for chamado, uma para cada chamada. O operador de restrição $(\nu n)P$ permite criar um novo ambiente (n) que será utilizado pelo processo P. O operador é chamado restrição porque somente o processo P saberá de sua existência, ele não poderá ser utilizado por outro processo. No Cálculo-π, esse operador é utilizado para criar novos canais de comunicação. A Figura 4.2 apresenta as convenções sintáticas do CA.

[4] Apesar da classe sintática de expressões M poder ser tanto um nome quanto uma capacidade, somente nomes são aceitos para nomear ambientes. Expressões que utilizam capacidades como nomes, por exemplo, $in\ n[P]$ são inválidas. Essa restrição pode ser imposta através de um sistema de tipos, conforme será descrito na Seção 4.2

Convenções sintáticas

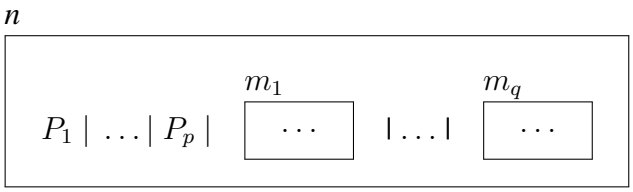
$(vn)P\|Q$	é lido como	$((vn)P)\|Q$
$!P\|Q$	é lido como	$(!P)\|Q$
$M.P\|Q$	é lido como	$(M.P)\|Q$
$(x).P\|Q$	é lido como	$((x).P)\|Q$
$(\nu n_1 \dots n_k)P$	$\triangleq$	$(\nu n_1) \dots (\nu n_k)P$
$n[]$	$\triangleq$	$n[\mathbf{0}]$
M	$\triangleq$	$M.\mathbf{0}$ (onde for apropriado)

Figura 4.2 – Convenções sintáticas do Cálculo de Ambientes

Um ambiente é uma estrutura em árvore com ambientes aninhados, representados por colchetes. Cada nó dessa árvore pode conter vários processos em paralelo e outros ambientes. Um ambiente possui o seguinte formato:

$$n[P_1| \ \dots \ | \ P_p \ | \ m_1[\dots] \ | \ \dots \ | \ m_q[\dots]] \quad (P_i \neq n_i[\dots])$$

Neste exemplo, n é um ambiente composto por processos e outros ambientes. $P_1| \ \dots \ | \ P_p$ representa os processos internos do ambiente n, onde $\dots$ significa que podem existir mais processos ou ambientes. Ou também pode-se representar o mesmo exemplo com retângulos aninhados:



As capacidades permitem que ambientes executem operações em outros ambientes. Por exemplo, o processo $M.P$ executa uma ação através da capacidade M e continua o processo como P. Enquanto a ação M estiver sendo executada o processo P é interrompido.

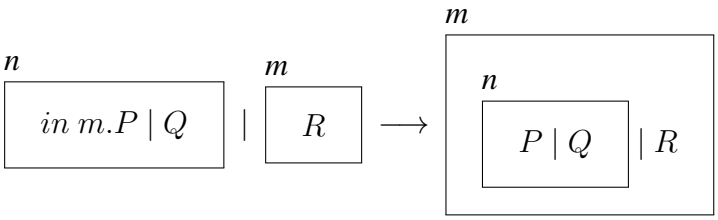
A capacidade in pode ser utilizada na ação:

$$in \ m.P$$

que informa ao ambiente onde estiver sendo exercida a capacidade para entrar no ambiente m mais próximo e seguir executando P. A regra de redução da capacidade in é a seguinte:

$$n[in \ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R] \qquad \text{(Redução } \textit{in}\text{)}$$

Ou visualizando os ambientes em retângulos:



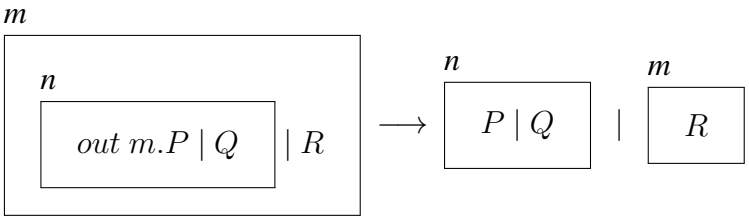
A capacidade out pode ser utilizada na ação:

$$out\ m.P$$

que informa ao ambiente onde estiver sendo exercida a capacidade para sair de seu ambiente superior (pai) e seguir executando P. Nesse exemplo o ambiente superior é o m. A regra de redução da capacidade out é a seguinte:

$$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R] \qquad \text{(Redução } \textit{out}\text{)}$$

Ou visualizando os ambientes em retângulos:



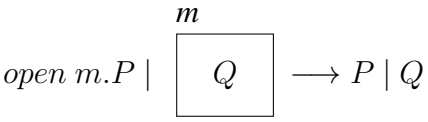
Por fim, a capacidade $open$ pode ser utilizada na ação:

$$open\ m.P$$

que elimina o ambiente m, fazendo com que o conteúdo do mesmo, no caso P, seja executado fora do ambiente, conforme a regra de redução:

$$open\ m.P \mid m[Q] \rightarrow P \mid Q \qquad \text{(Redução } \textit{open}\text{)}$$

Ou visualizando os ambientes em retângulos:



As entidades que podem ser utilizadas para comunicação são nomes ou capacidades (classe sintática de expressões M). A forma de comunicação mais simples é anônima e local

dentro de um ambiente. Uma ação de saída ($\langle M \rangle$) envia como mensagem uma capacidade ou nome dentro do ambiente. Uma ação de entrada ($(x).P$) captura a mensagem e efetua a substituição da variável pela mensagem. Essa sequência de passos é ilustrada através da seguinte redução (sendo que $P\{x \leftarrow M\}$ significa a substituição de M por cada ocorrência de x no processo P):

$$(x).P \mid \langle M \rangle \longrightarrow P\{x \leftarrow M\} \qquad \text{(Redução } \textit{com}\text{)}$$

Ambientes podem ser utilizados para modelar máquinas e pacotes, ou seja, uma troca de mensagens entre máquinas distintas (GORDON, 1998).

$$\overbrace{A[\underbrace{msg[out\ A.in\ B \mid \langle M \rangle]}_{\text{A} \rightarrow \text{B: M}}]}^{\text{Máquina A}} \mid \overbrace{B[\underbrace{open\ msg.(x).P}_{\text{receba } x;\ P}]}^{\text{Máquina B}} \qquad \text{(pacotes)}$$

É possível avaliar a expressão utilizando as regras de reduções in, out, $open$ e com descritas nesta seção:

$$\begin{aligned}
& & A[msg[out\ A.in\ B \mid \langle M \rangle]] \mid B[open\ msg.(x).P] \\
\text{(Redução } \textit{out}\text{)} \quad & \rightarrow & A[] \mid msg[in\ B \mid \langle M \rangle] \mid B[open\ msg.(x).P] \\
\text{(Redução } \textit{in}\text{)} \quad & \rightarrow & A[] \mid B[msg[\langle M \rangle] \mid open\ msg.(x).P] \\
\text{(Redução } \textit{open}\text{)} \quad & \rightarrow & A[] \mid B[\langle M \rangle \mid (x).P] \\
\text{(Redução } \textit{com}\text{)} \quad & \rightarrow & A[] \mid B[P\{x \leftarrow M\}]
\end{aligned}$$

Na Figura 4.3 é visualizada a congruência estrutural do CA. A congruência estrutural demostra quando duas expressões são estruturalmente iguais, ou seja, define classes de equivalências entre expressões. Na matemática, por exemplo, as expressões abaixo são estruturalmente iguais:

$$1 + 2 \times 3 \quad \equiv \quad 1 + (2 \times 3)$$

O CA pode ser utilizado como base para linguagens de programação para a computação móvel, tornando o código mais simples de entender e de manter (WEIS; BECKER; BRANDLE, 2006).

Congruência Estrutural

$P \equiv P$		(Estrut Refl)
$P \equiv Q \Rightarrow Q \equiv P$		(Estrut Simm)
$P \equiv Q,\ Q \equiv R \Rightarrow P \equiv R$		(Estrut Trans)
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$		(Estrut Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$		(Estrut Par)
$P \equiv Q \Rightarrow !P \equiv !Q$		(Estrut Repl)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$		(Estrut Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$		(Estrut Ação)
$P \equiv Q \Rightarrow (n).P \equiv (n).Q$		(Estrut Entrada)
$P \mid Q \equiv Q \mid P$		(Estrut Par Com)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$		(Estrut Par Ass)
$!P \equiv P \mid !P$		(Estrut Repl Par)
$(\nu n, m)P \equiv (\nu m, n)P$	$se\ n \neq m$	(Estrut Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$	$se\ n \notin fn(P)$	(Estrut Res Par)
$(\nu n)m[P] \equiv m[(\nu n)P]$	$se\ n \neq m$	(Estrut Res Amb)
$P \mid 0 \equiv P$		(Estrut Zero Par)
$(\nu n)0 \equiv 0$		(Estrut Zero Res)
$!0 \equiv 0$		(Estrut Zero Repl)
$\varepsilon.P \equiv P$		(Estrut ε)
$(M.M').P \equiv M.M'.P$		(Estrut .)

Figura 4.3 – Congruência Estrutural do Cálculo de Ambientes

4.2 Cálculo de Ambientes Tipado

Existem várias propostas na literatura para adicionar tipos ao CA. Alguns sistemas de tipos visam controlar a comunicação, a mobilidade, a segurança ou ambos. Dois sistemas de tipos propostos para o CA serão descritos nas próximas seções. Os dois sistemas foram propostos pelo autores do CA, sendo que o foco do primeiro sistema é controlar a comunicação entre processos, ou seja, permitir a troca de mensagens somente se os tipos do processo emissor e receptor combinarem. No segundo sistema, além da comunicação o foco é o controle da mobilidade. Dessa forma, existem ambientes que podem ser movidos ou não. O sistema deve garantir, por exemplo, que as capacidades in e out sejam executadas somente em ambientes móveis.

4.2.1 Sistema de tipos para comunicação

O primeiro sistema de tipos proposto para o CA foi proposto por Cardelli e Gordon (1999). O principal foco desse sistema de tipos é garantir que a comunicação sempre seja efetuada de forma apropriada. Mensagens trocadas dentro de ambientes não são direcionadas.

Dessa forma, não existe um controle se um processo está recebendo uma mensagem que ele saiba interpretar. Com essa motivação, é proposto um sistema de tipos para adicionar anotações de tipos em processos e em mensagens. Com essas anotações de tipos é possível conhecer previamente que tipo de mensagem um processo espera, garantindo que a comunicação seja efetuada de forma correta entre mensageiro e receptor. Nesse sistema de tipos não é tratada a questão da mobilidade. Na Figura 4.4 é visualizado os tipos propostos.

Tipos

$W ::=$	tipos para mensagens
$Amb[T]$	nome de ambiente que permite troca T
$Cap[T]$	capacidade que desencadeia trocas de T
$S, T ::=$	tipos para troca
Shh	sem troca
$W_1 \times \ldots \times W_k$	tupla de troca

Figura 4.4 – Tipos para controlar a comunicação

Tipos de mensagens podem ser $Amb[T]$, ou seja, o tipo para nomes de ambientes que internamente permitem trocas com tipos T e $Cap[T]$, ou seja, o tipo para capacidades que quando utilizadas podem desencadear trocas de T. Os tipos de troca podem ser Shh (ausência de troca de mensagens) ou $W_1 \times \ldots \times W_k$, uma tupla de mensagens com seus respectivos tipos. Para $k = 0$, a tupla vazia, é denominada simplesmente 1, ou seja, permite sincronização pura. Sincronização pura é uma forma de comunicação na qual os processos não se comunicam através da passagem de valores, somente através da sincronização de ações ou de eventos (HENNESSY; INGÓLFSDÓTTIR, 1993). Um ambiente que permite pura sincronização possui o tipo $Amb[1]$. Alguns exemplos de tipos que podem ser utilizados são $Amb[Shh]$, ou seja, um ambiente que não efetua trocas de mensagens, $Cap[Shh]$, uma capacidade que não desencadeia troca de mensagens e $Amb[Cap[Shh]]$, um ambiente que troca mensagens de capacidades do tipo Shh.

As regras de tipos são formalizadas na Figura 4.5. As sentenças válidas são:

- $E \vdash \diamond$ Ambiente estático de tipos [5] bem formado ou construído.
- $E \vdash M : W$ Expressão bem construída com tipo de mensagem W.
- $E \vdash P : T$ Processo bem construído com tipo de troca T.

[5] Não confundir "Ambiente" estático de tipos descrito na Seção 3.1 com "Ambiente" do Cálculo de Ambientes.

Regras de Tipos

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env n)
$$\frac{E \vdash \diamond \quad n \notin dom(E)}{E, n\text{: } W \vdash \diamond}$$

(Exp n)
$$\frac{E', n\text{: } W, E'' \vdash \diamond}{E', n\text{: } W, E'' \vdash n : W}$$

(Exp ε)
$$\frac{E \vdash \diamond}{E \vdash \varepsilon : Cap[T]}$$

(Exp .)
$$\frac{E \vdash M : Cap[T] \quad E \vdash M' : Cap[T]}{E \vdash M.M' : Cap[T]}$$

(Exp in)
$$\frac{E \vdash M : Amb[S]}{E \vdash in\ M : Cap[T]}$$

(Exp out)
$$\frac{E \vdash M : Amb[S]}{E \vdash out\ M : Cap[T]}$$

(Exp $open$)
$$\frac{E \vdash M : Amb[T]}{E \vdash open\ M : Cap[T]}$$

(Proc Ação)
$$\frac{E \vdash M : Cap[T] \quad E \vdash P : T}{E \vdash M.P : T}$$

(Proc Amb)
$$\frac{E \vdash M : Amb[T] \quad E \vdash P : T}{E \vdash M[P] : S}$$

(Proc Res)
$$\frac{E, n : Amb[T] \vdash P : S}{E \vdash (\nu n\text{: } Amb[T])\ P : S}$$

(Proc Zero)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{0} : T}$$

(Proc Par)
$$\frac{E \vdash P : T \quad E \vdash Q : T}{E \vdash P \mid Q : T}$$

(Proc Repl)
$$\frac{E \vdash P : T}{E \vdash !P : T}$$

(Proc Entrada)
$$\frac{E, n_1\text{:}W_1, \ \ldots\ , n_k\text{:}W_k \vdash P : W_1 \times \ldots \times W_k}{E \vdash (n_1\text{:}W_1, \ \ldots\ , n_k\text{:}W_k).P : W_1 \times \ldots \times W_k}$$

(Proc Saída)
$$\frac{E \vdash M_1 : W_1 \quad \ldots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \ \ldots\ , M_k \rangle : W_1 \times \ldots \times W_k}$$

Figura 4.5 – Sistema de tipos proposto em (CARDELLI; GORDON, 1999)

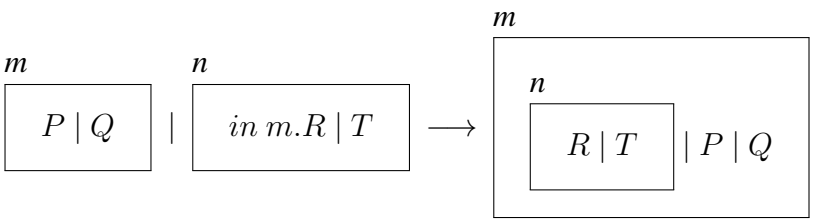
As três primeiras regras são básicas para a construção do restante do sistema de tipos. A regra (Env $\varnothing$) somente informa que um ambiente vazio é bem construído; (Env n) diz que n do tipo W é bem construído em E, desde que n ainda não esteja declarado em E; (Exp n) informa que se a concatenação de um ambiente E' (que já possui na sua lista n do tipo W) com outro ambiente E'' é bem construída [6], então nesse novo ambiente resultante da concatenação, n continuará sendo do tipo W.

Em (Exp ε) a regra fixa o tipo $Cap[T]$ para caminhos vazios ou nulos. Em (Exp .) a regra diz que se existir duas capacidades com o mesmo tipo, as duas podem ser combinadas em um caminho, e o tipo dessa nova expressão continuará o mesmo das capacidades.

Com relação às capacidades, as regras de tipos para in (Exp in) e out (Exp out) são as mesmas. Elas informam que se um ambiente possui tipo de troca S ($Amb[S]$) então a capacidade in ou out possuirão o tipo $Cap[T]$ quando aplicadas nesse ambiente. O importante dessas regras é que o tipo da capacidade não precisa ser o mesmo tipo do ambiente, visto que in e out não desencadeiam trocas de tipos, somente movem ambientes para dentro ou fora de

[6] Conforme Andrew Gordon (1994), a concatenação de ambientes de tipos é bem construída se $dom(E') \cap dom(E'') = \varnothing$

outros ambientes. Como a troca de mensagens somente é permitida dentro de ambientes, o tipo do ambiente a ser movido não precisa coincidir com o que receberá o ambiente. Em contrapartida, a capacidade $open$ quando aplicada a um ambiente faz com que ele seja dissolvido e seu conteúdo seja liberado, ou seja, as trocas de mensagens que eram executadas dentro do ambiente agora são executadas fora dele (no ambiente superior). Dessa forma, o tipo do ambiente deve combinar com o tipo da capacidade. Essa regra é descrita em (Exp $open$). Por exemplo, considere dois ambientes em paralelo: $m : Amb[S]$ e $n : Amb[T]$. Apesar de os tipos de troca serem diferentes, n deseja entrar em m e, dessa forma, torna-se filho do mesmo:



Conforme exemplificado, a capacidade in foi executada com sucesso, e apesar de n estar dentro de m os processos R e T só podem se comunicar entre si. Por essa razão, não é necessário se preocupar com os tipos dos ambientes para as capacidades in e out, pois elas movem os ambientes inteiros e não processos isolados. A forma de comunicação utilizada pelo CA pode ser vista como uma desvantagem, pois sempre que for necessário se comunicar com um processo, primeiramente deve-se abrir o ambiente (e nesse caso o ambiente é destruído) para só depois liberar os processos que estavam contidos dentro do ambiente. Com essa restrição, é bastante complexo o trabalho de modelar um exemplo do mundo real utilizando ambientes, como por exemplo, uma casa inteligente. Sempre que um aparelho quisesse se comunicar com outro seria necessário criar *buffers* ou canais específicos para comunicação, fazendo com que os processos depositassem e abrissem as suas mensagens em um local centralizado. Essa restrição é tratada em uma variante do CA, chamada *Boxed Ambients*, que será descrita na Seção 4.3.

A regra (Proc Ação) restringe que uma ação só pode ser executada se a capacidade possuir o tipo $Cap[T]$ e se o processo a ser executado após a capacidade ser exercida também possuir tipo de troca T. Em (Proc Amb), a regra restringe que um processo P com tipo de troca T só poderá ser executado dentro de um ambiente se o ambiente permitir trocas T, ou seja, possuir o tipo $Amb[T]$. Um ambiente não efetua troca de mensagens (somente seus processos internos), por isso na regra é atribuído o tipo S ao processo $M[P]$. Para cada nome de ambiente é dado o tipo $Amb[T]$, para o *typechecker* poder inferir que qualquer mensagem recebida por um processo dentro desse ambiente possui o tipo T. Porém o tipo de troca externa do ambiente

é S, pois um ambiente não troca mensagens, somente os processos internamente.

O processo **0** não efetua nenhuma troca de mensagem, pois é um processo que demonstra inatividade, assim ele possui naturalmente o tipo Shh. Porém, a regra (Proc Zero) convenciona que **0** pode ter qualquer tipo T. Dessa forma, **0** pode ser colocado em paralelo com qualquer outro processo, por exemplo: $P : T \mid \mathbf{0} : T$. Com esse exemplo, a regra (Proc Par) também é explicada. Somente é possível colocar dois processos em paralelo se eles possuírem o mesmo tipo de troca. A regra (Proc Repl) diz que se um processo possui tipo de troca T, então a replicação desse processo continuará com o mesmo tipo T. A regra (Proc Res) permite que a qualquer momento seja adicionada uma variável do tipo $Amb[T]$ em E, e após essa inclusão, seja criado um novo ambiente para uso exclusivo de P, através do operador ν.

Por fim, em relação à comunicação, a regra (Proc Entrada) convenciona que a qualquer momento podem ser adicionadas uma ou mais variáveis do tipo W em E, e caso exista um processo com o mesmo tipo de troca das novas variáveis, é possível montar uma expressão para entrada de dados. A regra (Proc Saída) convenciona que se existe uma ou mais mensagens do tipo W, o envio dessa mensagem no ambiente possuirá o mesmo tipo de troca W.

Com essas regras, é possível tipar o exemplo dos pacotes descrito na Seção 4.1:

$$A[msg[\underbrace{out\ A.in\ B}_{Cap[W]} \mid \langle M \rangle]] \mid B[\underbrace{open\ msg}_{Cap[W]}.(x : W).P] : Shh \qquad \text{(pacote tipado)}$$

Para o exemplo ser bem tipado, deve-se conhecer previamente que $A : Amb[Shh]$, $B : Amb[W]$, $msg : Amb[W]$, $M : W$, e $P : W$. Na Figura A.1 do Apêndice A pode ser visualizada a árvore de derivação dessa expressão, provando que ela é bem tipada. A mesma prova não pode ser concluída se $B : Amb[Shh]$, conforme pode ser visualizado na Figura A.2. A prova não pode ser concluída porque o ambiente B precisa permitir trocas do tipo W, ou seja, $Amb[W]$, para conseguir abrir o ambiente $msg : Amb[W]$ e receber a mensagem.

4.2.2 Sistema de tipos para mobilidade

Outro sistema de tipos foi obtido através da extensão do sistema de tipos para controle de comunicação (CARDELLI; GORDON, 1999), descrito na seção anterior. Nesse novo sistema o foco é o controle da mobilidade, através do conceito de “imobilidade” e de “travas” (CARDELLI; GORDON; GHELLI, 1999). A principal ideia desse novo sistema de tipos é controlar a mobilidade dos ambientes, restringindo quando um ambiente pode se mover e quando ele pode ser aberto. Na Figura 4.6 podem ser visualizados os novos tipos.

Tipos

$Y ::=$	anotações de travas
$\bullet$	travado
$\circ$	destravado
$Z ::=$	anotações de mobilidade
$\veebar$	imóvel
$\curvearrowright$	móvel
$W ::=$	tipos para mensagens
$Amb^{Y}[^{Z}T]$	nome de ambiente que permite troca T
$Cap[^{Z}T]$	capacidade que desencadeia trocas de T
$T ::=$	tipos para troca
Shh	sem troca
$W_1 \times \ldots \times W_k$	tupla de troca

Figura 4.6 – Tipos para controlar a mobilidade

A única diferença para os tipos apresentados na Seção 4.2.1 é a adição das anotações de travas e mobilidades. Um ambiente pode ser considerado como travado ($^{\bullet}$) ou destravado ($^{\circ}$). Ambientes travados não podem ser abertos e destravados podem ser abertos através da capacidade $open$. A anotação de trava é adicionada ao tipo do ambiente. A segunda adição feita ao sistema de tipos são as anotações de mobilidade. A ideia é controlar a comunicação e a mobilidade dos processos, adicionado um atributo para informar que um processo, além do seu tipo (T), também pode executar movimento (atributo $\curvearrowright$) ou não (atributo $\veebar$). As capacidades in e out são exemplos de capacidades que executam movimento ($Cap[\curvearrowright T]$).

As regras de tipos são iguais às do sistema anterior, conforme pode ser visualizado na Figura 4.7, bastando adicionar as anotações de travas e de mobilidades onde for necessário. As novas regras asseguram que cada comunicação deve ser bem tipada, que nenhum ambiente travado poderá ser aberto, e que as capacidades in e out não poderão ser executadas em ambientes imóveis.

Assim como apresentado na Seção 4.2.1, é possível provar que a expressão pacote tipado

$$A[msg[out\ A.inB \mid \langle M \rangle]] \mid B[open\ msg.(x : W).P] : {}^{\veebar}Shh$$

é bem tipada, desde que A seja um ambiente travado e imóvel ($Amb^{\bullet}[^{\veebar}Shh]$), msg um ambiente destravado e móvel ($Amb^{\circ}[\curvearrowright W]$) e B um ambiente travado e móvel ($Amb^{\bullet}[\curvearrowright W]$). Além disso, P deve ter tipo $\curvearrowright W$. A derivação é provada na Figura A.3 do Apêndice A.

Regras de Tipos

(Env $\varnothing$) $\dfrac{}{\varnothing \vdash \diamond}$ (Env n) $\dfrac{E \vdash \diamond \quad n \notin dom(E)}{E, n\colon W \vdash \diamond}$ (Exp n) $\dfrac{E', n\colon W, E'' \vdash \diamond}{E', n\colon W, E'' \vdash n : W}$ (Exp ε) $\dfrac{E \vdash \diamond}{E \vdash \varepsilon : Cap[^{Z}T]}$

(Exp .) $\dfrac{E \vdash M : Cap[^{Z}T] \quad E \vdash M' : Cap[^{Z}T]}{E \vdash M.M' : Cap[^{Z}T]}$ (Proc Repl) $\dfrac{E \vdash P : {}^{Z}T}{E \vdash !P : {}^{Z}T}$

(Exp in) $\dfrac{E \vdash n : Amb^{Y}[^{Z}T]}{E \vdash in\ n : Cap[^{\frown}T']}$ (Exp out) $\dfrac{E \vdash n : Amb^{Y}[^{Z}T]}{E \vdash out\ n : Cap[^{\frown}T']}$ (Exp $open$) $\dfrac{E \vdash n : Amb^{\circ}[^{Z}T]}{E \vdash open\ n : Cap[^{Z}T]}$

(Proc Ação) $\dfrac{E \vdash M : Cap[^{Z}T] \quad E \vdash P : {}^{Z}T}{E \vdash M.P : {}^{Z}T}$ (Proc Amb) $\dfrac{E \vdash M : Amb^{Y}[^{Z}T] \quad E \vdash P : {}^{Z}T}{E \vdash M[P] : {}^{Z'}T'}$

(Proc Res) $\dfrac{E, n : Amb^{Y}[^{Z}T] \vdash P : {}^{Z'}T'}{E \vdash (\nu n{:}Amb^{Y}[^{Z}T])\ P : {}^{Z'}T'}$ (Proc Zero) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{0} : {}^{Z}T}$ (Proc Par) $\dfrac{E \vdash P : {}^{Z}T \quad E \vdash Q : {}^{Z}T}{E \vdash P \mid Q : {}^{Z}T}$

(Proc Entrada) $\dfrac{E, n_1\colon W_1, \ \ldots\ , n_k\colon W_k \vdash P : {}^{Z}W_1 \times \ldots \times W_k}{E \vdash (n_1\colon W_1, \ \ldots\ , n_k\colon W_k).P : {}^{Z}W_1 \times \ldots \times W_k}$ (Proc Saída) $\dfrac{E \vdash M_1 : W_1 \quad \ldots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \ \ldots\ , M_k \rangle : {}^{Z}W_1 \times \ldots \times W_k}$

Figura 4.7 – Sistema de tipos proposto em (CARDELLI; GORDON; GHELLI, 1999)

4.3 *Boxed Ambients*

Existem algumas variantes do CA encontradas na literatura. O objetivo dessas variantes é estender ou alterar alguma funcionalidade do CA puro, com os mais diferentes focos: aumentar a capacidade de comunicação, flexibilidade, segurança etc. Alguns exemplos são os Ambientes Seguros (*Safe Ambients*) (LEVI; SANGIORGI, 2003), Cálculo M^3 (COPPO et al., 2003), *Boxed Ambients* (BUGLIESI; CASTAGNA; CRAFA, 2004), Ambientes lógicos (SANGIORGI, 2001), *Finite-Control Mobile Ambients* (CHARATONIK; GORDON; TALBOT, 2002) entre outros. Um artigo que descreve as principais variantes do CA no quesito segurança pode ser lido em (MARGARIA; ZACCHI, 2008). Será descrito nesta seção com maiores detalhes somente o *Boxed Ambients* (BA), uma variante do CA que foi utilizada no Cálculo de Ambientes Sensível ao Contexto, que será descrito na Seção 4.4.

Conforme inicialmente abordada na Seção 4.2.1, a forma de comunicação do CA possui

algumas restrições. A comunicação ocorre de forma anônima e somente dentro dos ambientes. Se dois processos estão em diferentes ambientes e desejam se comunicar entre si, um dos ambientes deve ser destruído através da capacidade $open$, fazendo assim com que o processo seja liberado e executado fora do ambiente. Por exemplo, um processo P rodando dentro de uma máquina h deseja acessar um programa Q que está em uma outra máquina. Para fazer a cópia, um ambiente intermediário a é criado que contém o programa Q. Essa situação pode ser modelada da seguinte forma: $a[in\ h.Q] \mid h[P]$. Quando a cópia estiver completa, o novo estado do ambiente será modelado como: $h[a[Q] \mid P]$. O programa Q pode executar normalmente dentro de a porém, não pode interagir com P pois estão em ambientes diferentes. Para permitir a interação, o ambiente a deve ser destruído, liberando o processo Q através da capacidade $open$: $h[a[Q] \mid open\ a.P] \rightarrow h[Q \mid P]$. Outra restrição é que, apesar de agora os processos poderem interagir, não é possível saber o que Q deve fazer e nem com quem. Além disso, a capacidade $open$ pode ser um risco de segurança para sistemas distribuídos, pois a poderia ser um ambiente mal-intencionado carregando um vírus, e quando fosse aberto, tal vírus seria espalhado dentro do ambiente que executou a capacidade. Essas restrições motivaram a criação do BA (BUGLIESI; CASTAGNA; CRAFA, 2004).

O BA herda do CA as primitivas in e out, mas não $open$. Para o cálculo não perder expressividade, são acrescentadas novas primitivas para permitir a comunicação direta entre ambiente pai e filho. Ao invés de abrir um ambiente para liberar uma mensagem, a mensagem é direcionada diretamente para um ambiente. Porém, a comunicação dentro dos ambientes continua sendo anônima, pois não se define qual processo deve receber a mensagem. A mensagem somente é liberada dentro do ambiente. Um BA, de uma forma resumida, é um ambiente móvel que não pode ser aberto. Na Figura 4.8 pode ser visualizada a sintaxe do BA.

A diferença de sintaxe do BA para o CA é que não existe a capacidade $open$, e na comunicação pode ser informada a localização onde será realizada a comunicação, representada por η. No caso de omissão da localização, a comunicação será executada localmente. Nesse caso, o operador $\star$ não é opcional. A tupla de mensagens $x_1, \ldots, x_k$ pode ser abreviada para $\tilde{x}$. Com essa sintaxe, pode ser modelado um processo que espera uma mensagem do ambiente n, ou seja, explicitando o nome, ou simplesmente dizendo que espera uma mensagem de um ambiente superior. Estes processos podem ser visualizados nos exemplos:

$$(\tilde{x})^n P \mid n[\langle \tilde{M} \rangle Q \mid R] \qquad \text{(Espera mensagem de } n\text{)}$$

$$n[(\tilde{x})^{\uparrow} P \mid Q] \mid \langle \tilde{M} \rangle R \qquad \text{(Espera mensagem de } \uparrow\text{)}$$

Sintaxe

$M ::=$	expressões
$a - q$	nomes
$x - z$	variáveis
$in\ M$	entrar em ***M***
$out\ M$	sair de ***M***
$M.M$	caminho
$\eta ::=$	localizações
M	filho
$\uparrow$	pai
$\star$	local
$P ::=$	processos
$\mathbf{0}$	parado
$M.P$	ação
$(\nu n)P$	restrição
$P \mid P$	composição
$M[P]$	ambiente
$!P$	replicação
$(x_1, \ldots, x_k)^\eta P$	entrada, $k \geqslant 0$
$\langle M_1, \ldots, Mk \rangle^\eta P$	saída, $k \geqslant 0$

Figura 4.8 – Sintaxe do *Boxed* Ambientes

4.4 Sensibilidade ao Contexto no Cálculo de Ambientes

O Cálculo de Ambientes Sensível ao Contexto (CASC) (SIEWE; CAU; ZEDAN, 2009; SIEWE; ZEDAN; CAU, 2011), é uma extensão do CA e do BA. A principal ideia dos autores dessa extensão é criar novas funcionalidades no CA que permitam expressar e modelar contextos. Na Figura 4.9 é apresentada a sintaxe dos processos e das capacidades do CASC.

Os cinco primeiros processos (inatividade até replicação) são herdados do CA. A "ação sensível ao contexto" M, representada por $k?M.P$ é uma ação que executa M e continua como o processo P, somente se o contexto k for satisfeito. Por exemplo, considere um telefone celular que consegue detectar o ambiente no qual está inserido através da operação **user_at** e possui a capacidade **switchto** que permite alterar o estilo do toque telefônico. Deseja-se que o telefone troque automaticamente o estilo de toque para "silencioso" se o usuário estiver na sala de conferências. Mapeando o telefone para um ambiente, sendo que o usuário será o seu ambiente pai, é possível modelar uma ação sensível ao contexto:

phone

$$!\underbrace{\mathbf{user\_at}(conf)}_{k}?\underbrace{\mathbf{switchto}\langle silent\rangle}_{M}.\underbrace{\mathbf{0}}_{P}$$

Sintaxe

$P, Q ::=$	processos
$\mathbf{0}$	inatividade
$P \mid Q$	composição de processos
$(\nu n)P$	restrição
$n[P]$	ambiente
$!P$	replicação
$k?M.P$	ação sensível ao contexto
$x \triangleright (\tilde{y}).P$	abstração de processo
$\alpha ::=$	localizações
$\uparrow$	qualquer superior (pai)
$n \uparrow$	pai n
$\downarrow$	qualquer inferior (filho)
$n \downarrow$	filho n
$::$	qualquer irmão
$n ::$	irmão n
ε	localmente
$M ::=$	capacidades
$del\ n$	apagar *n*
$in\ n$	entrar em *n*
out	sair do ambiente
$\alpha\ x\langle\tilde{y}\rangle$	chamada de processo
$\alpha\ (\tilde{y})$	entrada
$\alpha\ \langle\tilde{y}\rangle$	saída

Figura 4.9 – Sintaxe dos processos e das capacidades do CASC

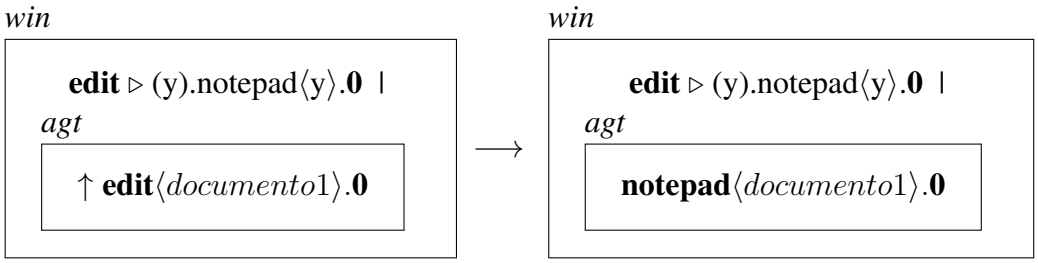
O ambiente telefone (*phone*) deverá avaliar a todo o momento a expressão **user_at**$(conf)$ para verificar se o usuário está na sala de conferência e assim executar a capacidade. Essa é uma das formas através das quais o CASC possibilita a aquisição de contextos. A segunda forma é através de abstrações de processos, representada pela sintaxe $x \triangleright (\tilde{y}).P$. Nessa sintaxe, o nome x é vinculado ao processo P, sendo que $\tilde{y}$ é a lista de parâmetros do processo. Conforme visualizado na Figura 4.9, uma invocação dessa abstração possui a sintaxe $\alpha\ x\langle\tilde{z}\rangle$, sendo que $\tilde{z}$ é a lista de parâmetros a serem passados. Para exemplificar, considere um agente denominado agt utilizado para editar um arquivo texto. O agente foi acionado para editar o arquivo *documento1*. Modelando o agente como um ambiente e uma abstração de processo $edit$, temos:

agt

$\uparrow$ **edit**$\langle documento1\rangle.\mathbf{0}$

O operador $\uparrow$ indica que $edit$ está definido em algum ambiente superior a agt. O usuário

migra o seu trabalho para uma máquina que possui um Sistema Operacional (SO) *Windows*. Nesse sistema, o processo utilizado para editar arquivos de texto é o bloco de notas (*notepad.exe*). Dessa forma, uma requisição ao agente para editar um arquivo será executada de acordo com a redução:



Se o SO utilizado fosse baseado em alguma distribuição *Linux*, $edit$ poderia ser configurado para utilizar o editor *emacs* ou o *vi*. Dessa forma, uma chamada de processo no formato $\alpha\ x\langle\tilde{y}\rangle$ pode se comportar de diferentes formas dependendo da localização do ambiente que executar a chamada. Por esse motivo, uma chamada de processo é considerada uma forma de aquisição de contexto no CASC, enquanto que uma abstração de processo é uma forma de provimento de contexto.

Com relação às capacidades, a capacidade in é igual à capacidade in do CA. Porém, no CASC a capacidade out não possui parâmetros. O ambiente que executa a capacidade $open$ simplesmente move-se para fora de seu ambiente superior. Os autores do CASC justificam a não utilização da capacidade $open$ por trazer riscos de segurança, conforme descrito na Seção 4.3. A alternativa utilizada é a nova capacidade del, que elimina ambientes que já executaram seus processos (exemplo $n[\mathbf{0}]$), ou seja, del somente pode abrir (e eliminar) ambientes vazios.

Ambientes podem trocar mensagens através da capacidade $\alpha\ \langle\tilde{y}\rangle$ que envia uma lista de nomes $\tilde{y}$ para o ambiente localizado em α. Similarmente, a capacidade $\alpha\ (\tilde{y})$ é utilizada para receber uma lista de nomes do ambiente localizado em α.

4.4.1 Contextos

Conforme descrito na seção anterior, o CASC estende o CA. Dessa forma, as entidades utilizadas para representar uma sistema sensível ao contexto como, por exemplo, usuário, localização, sensores, etc, são modeladas com o conceito de ambientes. Devido à natureza móvel dos ambientes, a estrutura hierárquica dos ambientes se modifica conforme os processos são executados. Nessa estrutura, o contexto de um subprocesso é obtido substituindo-se o subprocesso por um espaço reservado $\odot$. Por exemplo, no sistema modelado por $P \mid n[Q \mid m[R \mid S]]$,

o contexto do processo R é $P \mid n[Q \mid m[\odot \mid S]]$ e do ambiente m é $P \mid n[Q \mid \odot]$. Na Figura 4.10 é apresentada a sintaxe utilizada utilizada para modelar contextos no CASC.

Sintaxe

$C ::=$	contexto
0	nulo
$\odot$	espaço reservado
$n[C]$	localização
$C \mid P$	composição em paralelo
$(\nu n)C$	restrição

Figura 4.10 – Sintaxe dos contextos

O contexto **0** é o contexto vazio ou nulo, ou seja, não possui nenhuma informação de contexto. O espaço reservado $\odot$ representa a posição do contexto dentro de um processo. O contexto $n[C]$ significa que o comportamento interno do ambiente n é descrito pelo contexto C. O contexto $C \mid P$ informa que o processo P está rodando em paralelo com o contexto C. Dessa forma, C é uma parte do contexto do processo P. Por fim, o contexto $(\nu n)C$ informa que o escopo do nome n é limitado ao contexto C.

A forma de avaliação de contextos é representada por $C_1(C_2)$, ou seja, $C_1\{\odot \leftarrow C_2\}$. Essa notação significa a substituição de C_2 em cada cada ocorrência de $\odot$ em C_1. Por exemplo, o contexto C_1 pode ser modelado como o contexto de algum dispositivo carregado por Bob enquanto ele está na sala de conferência com Alice:

$$C_1 \triangleq conf[P \mid bob[\odot] \mid alice[Q]] \qquad (C_1)$$

Dessa forma, é possível modelar um processo que demonstra Bob utilizando seu celular na sala de conferência com Alice:

$$C_1(cel[S]) \triangleq conf[P \mid bob[cel[S]] \mid alice[Q]] \qquad (C_1(cel[S]))$$

Com base nessa representação formal de contextos, os autores desenvolveram uma lógica modal para especificar expressões de contexto k. A lógica modal é utilizada para expressar diferentes "modos" ou "tipos" de verdade, ou seja, além de somente verdadeiro ou falso (HUTH; RYAN, 2008). Lógicas modais possuem conectivos que permitem expressar os diferentes tipos de verdade. Para descrever uma lógica modal, é fundamental definir o tipo de verdade que se deseja expressar, e modelar as propriedades necessárias para expressá-las. Para definir se uma fórmula na lógica modal é verdadeira, utiliza-se uma relação de satisfação ($\models$)

(HUTH; RYAN, 2008). Na Figura 4.11, é apresentada a sintaxe da lógica modal desenvolvida no CASC.

Sintaxe

$k ::=$	expressão de contexto
true	verdadeiro
$n = m$	nomes iguais
$\bullet$	espaço reservado
$\neg k$	negação
$k_1 \mid k_2$	composição em paralelo
$k_1 \wedge k_2$	conjunção
$n[k]$	localização
$\mathtt{new}(n, k)$	revelação
$\oplus k$	próximo nível
$\Diamond k$	algum nível
$\exists x.k$	quantificador existencial

Figura 4.11 – Sintaxe de expressão de contexto

Devido à complexidade da lógica e à limitação do tema desta dissertação, tal lógica não será detalhada, somente alguns conceitos básicos. A expressão de contexto **true** satisfaz todos os contextos, e pode ser denotada por $C \models$ **true**. Uma expressão de contexto na forma $n = m$ é útil para verificar quando duas mensagens são iguais. Por exemplo, n pode ser um sensor que monitora a temperatura de um ambiente, devolvendo os valores B (baixo), M (médio) ou A (alto). Uma expressão de contexto pode ser modelada para executar alguma ação especial quando a temperatura do ambiente for alta; nesse caso $C \models (n = A)$. A expressão de contexto $\bullet$ é satisfeita apenas por $\odot$. Em uma expressão de contexto, o símbolo $\bullet$ representa a posição do processo que está avaliando a expressão. Por exemplo, a expressão de contexto $possui(n)$ é satisfeita somente se o ambiente que avalia a expressão não possui um ambiente superior e contiver um ambiente n. Essa expressão de contexto pode ser formalizada, utilizando a sintaxe apresentada na Figura 4.11:

$$possui(n) \quad \triangleq \quad \oplus\,(\bullet \mid n[\textbf{true}] \mid \textbf{true})$$

Exemplo de utilização:

$$bob[\odot \mid cel[S]] \models possui(cel)$$

Nesse exemplo, o ambiente que avalia a expressão é bob. Nesse caso, o contexto $bob[\odot \mid cel[S]]$, satisfaz $possui(cel)$, pois bob não possui um ambiente superior e, possui um

ambiente interno chamado cel. O contexto $conf[bob[\odot \mid cel[S]]]$ não satisfaz $possui(cel)$, pois bob possui um ambiente superior ($conf$). Nesse caso, pode-se utilizar a expressão de contexto $\oplus$ para mover-se exatamente um nível abaixo na hierarquia ou $\diamond$ para qualquer nível:

$$conf[bob[\odot \mid cel[S]]] \models \oplus possui(cel)$$

4.5 Resumo do capítulo

Este capítulo descreveu o CA, um modelo formal utilizado para descrever sistemas móveis. Como a mobilidade de código é um requisito da computação pervasiva, descrita no Capítulo 2, o CA pode ser utilizado para descrever o comportamento de aplicações pervasivas. O CA possui muitas variantes, adicionando melhorias ao modelo original, como por exemplo, permitindo a comunicação entre ambientes diferentes dentro de uma hierarquia. Um modelo recente é o CASC, descrito na Seção 4.4. Esse modelo adiciona novas primitivas para permitir descrever contextos, uma das características da Computação Pervasiva, descrita na Seção 2.2. O CASC é um modelo recente, dessa forma não existe nenhuma variante, nem um sistema de tipos proposto para ele, ao contrário de para o CA. Será proposto, no próximo capítulo, um sistema de tipos para o CASC, visando o controle de comunicação, no estilo do sistema de tipos proposto na Seção 4.2.1.

5 SISTEMA DE TIPOS NO CASC

O objetivo inicial do sistema de tipos é restringir a comunicação entre ambientes, evitando que os ambientes recebam mensagens que não saibam interpretar, pois, conforme apresentado na Seção 4.2.1, a comunicação entre ambientes é assíncrona e não direcionada.

A ideia principal é inspirada no sistema de tipos apresentado na Seção 4.2.1: adicionar anotações de tipos em mensagens e em processos. Contudo, algumas diferenças importantes na sintaxe do CASC (Figura 4.9) em relação ao CA puro devem ser levadas em consideração, tais como:

- No CASC a troca de mensagens pode ser efetuada entre qualquer ambiente
- As capacidades não desencadeiam trocas de mensagens, pois no CASC não é possível abrir ambientes ($open$ não existe).
- A capacidade out não possui parâmetros, e existe uma capacidade nova para eliminar ambientes (del).
- Novas funcionalidades para permitir a sensibilidade ao contexto, e abstrações de processo.

Antes de iniciar a definição do sistema de tipos, será necessário efetuar uma alteração na sintaxe do CASC originalmente apresentada na Figura 4.9. A alteração será efetuada na parte de comunicação, pois no CASC as mensagens podem ser enviadas para qualquer ambiente, mesmo sem conhecer o nome do mesmo (categoria sintática α). A nova sintaxe sugerida não permite a troca de mensagens entre ambientes sem conhecer o nome do ambiente, ou seja, serão removidas as comunicações *qualquer superior* ($\uparrow$), *qualquer inferior* ($\downarrow$) e *qualquer irmão* (::). Esta alteração se justifica principalmente por questões de segurança. Ambientes que permitem receber mensagens de qualquer local podem ser alvos de ataques de ambientes rodando algum código malicioso. Uma mensagem em massa contendo algum código mal-intencionado pode ser enviada a todos os ambientes filhos, por exemplo. Além disso, Margaria e Zacchi (2008), afirmam, que a forma utilizada para segurança no CA é o conhecimento do nome do ambiente. Conhecendo o nome do ambiente, por exemplo, é possível utilizar as primitivas in e out. Dessa forma, para permitir a comunicação entre ambientes sem conhecer o nome, seria necessário construir uma relação de grupos no qual um determinado ambiente pode interagir, o que não é o foco deste trabalho. Outra questão se deve ao sistema de tipos: se não for especificado

o nome do ambiente para qual a comunicação será feita, o sistema de tipos não poderá testar estaticamente se a comunicação será segura, ou seja, testar se os tipos dos ambientes emissor e receptor combinam. Com base nesses argumentos, a Figura 5.1 apresenta a nova sintaxe do CASC.

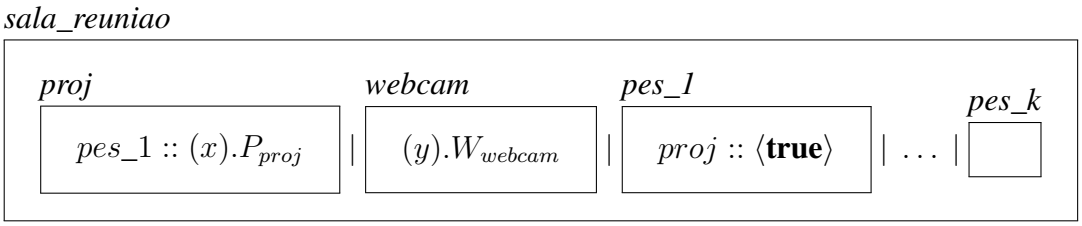
Sintaxe

$P, Q ::=$	processos
$\mathbf{0}$	inatividade
$P \mid Q$	composição de processos
$(\nu n : T)P$	restrição
$n[P]$	ambiente
$!P$	replicação
$k?M.P$	ação sensível ao contexto
$x : \textbf{\textit{Abs}}[T] \triangleright (\tilde{y}).P$	abstração de processo
$\alpha ::=$	localizações
$n \uparrow$	pai n
$n \downarrow$	filho n
$n ::$	irmão n
ε	localmente
$M ::=$	capacidades
$del\ n$	apagar ***n***
$in\ n$	entrar em ***n***
out	sair do ambiente
$\alpha\ x\langle\tilde{y}\rangle$	chamada de abstração de processo
$\alpha\ (\tilde{y})$	entrada
$\alpha\ \langle\tilde{y}\rangle$	saída

Figura 5.1 – Nova sintaxe dos processos e das capacidades do CASC

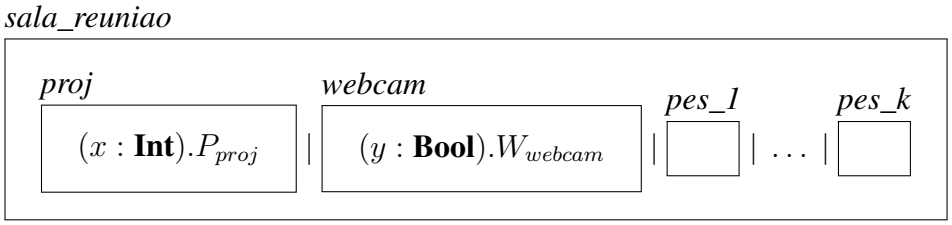
5.1 Sistema de tipos

Para justificar a necessidade de tipos, considere como exemplo uma sala de reuniões. A sala possui dois equipamentos eletrônicos, cada um modelado como um ambiente, P_{proj} que pode receber um parâmetro x (o número do *slide* de uma apresentação a ser mostrado) e W_{webcam} que recebe um parâmetro y (ligar ou desligar), além de várias pessoas que participam da reunião (*pes_1* até *pes_k*). Suponha que *pes_1* deseja interagir com o projetor (ambiente $proj$). Para isso ela envia o valor **true** para o mesmo. Tal cenário pode ser modelado da seguinte forma:



Sem um sistema de tipos, o processo P_{proj} receberá o valor **true** ($P_{proj}\{x \leftarrow \mathbf{true}\}$), porém não saberá como proceder, pois **true** não é um número de *slide* válido. Nesse caso, o comportamento do sistema será indefinido, podendo simplesmente não efetuar nenhuma ação, travar, ou lançar alguma exceção. Este é um exemplo para o uso de um sistema de tipos, conforme abordado no Capítulo 3 : prevenção de erros e segurança (página 23).

Com o uso de um sistema de tipos, o cenário e principalmente os processos podem ser modelados da seguinte maneira:



Analisando o exemplo e tomando como base o sistema de tipos apresentado na Seção 4.2.1, é possível sugerir anotações de tipos que impeçam a comunicação incorreta entre ambientes e processos. Primeiramente, é necessário conhecer previamente o tipo de dado de entrada ou de saída de um processo (o tipo de troca). No exemplo, é possível inferir que $(x : \mathbf{T}).P_{proj} : \mathbf{T}$ para algum **T**. Outro passo é adicionar um tipo para um ambiente, para conhecer previamente o tipo de dado de troca de um processo rodando dentro desse ambiente. No exemplo, o ambiente *proj* deve ter o tipo ***Amb***$[\mathbf{T}]$ para algum **T**.

Além de tipos básicos, a comunicação entre ambientes e processos pode ser feita através do envio de capacidades. Diferentemente do CA puro, no CASC a troca de mensagens (entrada e saída) também é tratada como uma capacidade (categoria sintática M, Figura 5.1). Isso se deve pela ação sensível ao contexto ($k?M.P$). Assim, a capacidade M só será executada se k for satisfeito, ou seja, a mensagem só será enviada ou recebida se o contexto k for satisfeito. Para diferenciar as capacidades, foi criado o tipo ***Cap***$[T]$. O T é o tipo propriamente dito que o processo envia ou recebe. Porém, o tipo T deve ficar dentro do ***Cap***$[]$ pois além do "comando" ser uma capacidade, ele possui o tipo de troca T. Dessa forma, no exemplo do projetor, o tipo do processo ficará como $(x : T).P_{proj} :$ ***Cap***$[T]$, consequentemente o ambiente que conterá esse processo possuirá o tipo ***Amb***$[$***Cap***$[T]]$.

Além dos tipos descritos, é necessário adicionar um tipo para diferenciar as abstrações de processo $x \triangleright (y : T).P : T$. Nesse caso, x, que é a abstração de processo, possuirá o tipo ***Abs***$[T]$, e o tipo de retorno será ***Cap***$[T]$, ou seja, $x :$ ***Abs***$[T] \triangleright (y : T).P :$ ***Cap***$[T]$. Na sintaxe do CASC, a chamada de uma abstração de processo também é tratada como uma capacidade. Nesse caso, a chamada da abstração possuirá o tipo ***Cap***$[T]$, ou seja, $x\langle M\rangle :$ ***Cap***$[T]$.

Conforme apresentado na Seção 4.4, as abstrações de processo são utilizadas para vincular um nome, por exemplo x, a um processo (ou a uma função). No exemplo apresentado em (SIEWE; ZEDAN; CAU, 2011), uma abstração de processo *edit* foi vinculada ao editor de textos *notepad* em um ambiente rodando no SO *Windows* e vinculada ao *emacs* em um SO *Linux*, conforme pode ser visualizado na Figura 5.2.

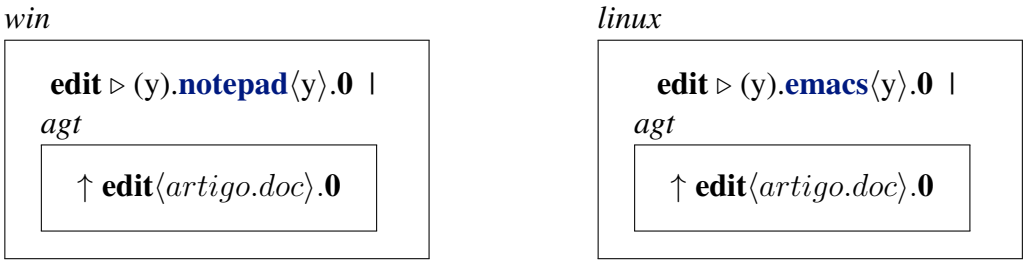


Figura 5.2 – Abstração de processo *edit* vinculada a editores de textos diferentes, conforme o ambiente (SIEWE; ZEDAN; CAU, 2011).

Dessa forma, quando invoca-se a abstração *edit*, o editor de texto é chamado, enviando juntamente uma mensagem com o nome do arquivo a ser editado: $edit\langle artigo.doc\rangle$. Essas explicações adicionais sobre abstrações de processo são importantes para entender a necessidade de criação do novo tipo ***Abs***$[T]$. Se as abstrações fossem tratadas somente por capacidades (***Cap***$[T]$), qualquer outra capacidade poderia ser invocada incorretamente como se fosse uma abstração, por exemplo: $in\langle artigo.doc\rangle$. Dessa forma, o tipo ***Abs***$[T]$ é essencial para o sistema de tipos, podendo assim existir uma regra de tipo restringindo que o comando seja usado de forma correta.

Outra ideia que surgiu durante a criação dos tipos para o CASC, foi a definição de subtipos (LISKOV; WING, 1994) para as capacidades. Nesse caso, as capacidades básicas (in, out, del) possuiriam o tipo ***Cap***$[T]$, as abstrações de processo o tipo ***Abs***$[T]$ e as entradas e saídas (comunicação) entre processos possuiriam o tipo ***Pro***$[T]$. Como qualquer uma das expressões descritas são capacidades e podem ser utilizadas na ação sensível ao contexto ($k?M.P$), então ***Abs***$[T]$ e ***Pro***$[T]$ seriam subtipos de ***Cap***$[T]$, ou seja, ***Abs***$[T] <:$ ***Cap***$[T]$ e ***Pro***$[T] <:$ ***Cap***$[T]$. Essa abordagem seria interessante para agrupar mais detalhadamente as expressões, ao invés

de todas serem tratadas genericamente como capacidades. O problema que inviabilizou essa abordagem, é que existe mais um tipo envolvido dentro dos colchetes, além da capacidade. Nesse caso, a relação de subtipos é dita somente pelo tipo mais externo, ou seja, ***Abs***$[T]$ é um subtipo de ***Cap***$[T]$, o que está correto. Porém pensando em uma linguagem de alto nível com tipos básicos como $bool$ e int, essa relação de subtipos permitiria dizer que ***Abs***$[bool]$ é um subtipo de ***Cap***$[int]$, o que não está correto. A solução para este problema, seria a definição de uma relação de subtipos para os tipos dentro dos []. Porém, nessa primeira versão do sistema de tipos, o foco é o controle de comunicação. Dessa forma, não é necessário detalhar todos os tipos da linguagem, o que dificulta uma construção mais exata da relação de subtipos.

Com base nesses argumentos, os tipos criados para o CASC podem ser visualizados na Figura 5.3.

Tipos

$W ::=$	tipos para mensagens
Amb$[T]$	ambiente que permite trocas T
Cap$[T]$	capacidade ou ação com tipo de troca T
Abs$[T]$	abstração de processo com tipo de troca T
$S, T ::=$	tipos para troca
Shh	sem troca
$W_1 \times \ldots \times W_k$	tupla de troca
$Bool ::=$	tipos para expressões de contexto

Figura 5.3 – Tipos para controlar a comunicação do CASC

As principais categorias de tipos são os tipos de mensagem W, tipos de troca T e tipos $Bool$, utilizados unicamente para tipar as expressões de contexto k. Os tipos de mensagem podem ser ***Amb***$[T]$, utilizados para tipar ambientes cujos processos rodando internamente possuem tipos de troca T (entrada e saída), ***Cap***$[T]$, utilizado para tipar as capacidades e ***Abs***$[T]$ para tipar as abstrações de processo. A importância de ter o tipo T em ***Amb***$[T]$, além de restringir a comunicação, é para o *typechecker* poder inferir o tipo correto para a entrada de um processo rodando dentro do ambiente. Por exemplo: $n[(x : \textbf{Tipo?}).P] :$ ***Amb***$[\textbf{Tipo?}]$. Os tipos de troca podem ser Shh, ou seja, ausência de comunicação, ou $W_1 \times \ldots \times W_k$, uma tupla de mensagens com vários elementos, cada um correspondendo a um tipo de mensagem W.

O sistema de tipos proposto é apresentado completamente na Figura 5.4. Nas regras de tipo, a notação $\tilde{n}$: $\tilde{W}$ é utilizada para abreviar n_1: W_1, $\ldots$, n_k:W_k. A notação $\tilde{W}\times$ abrevia $W_1 \times \ldots \times W_k$.

As regras de tipo (Env $\varnothing$), (Env n), (Exp n), (Proc Zero), (Proc Par), (Proc Amb),

Regras de Tipos

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env n)
$$\frac{\Gamma \vdash \diamond \quad n \notin dom(\Gamma)}{\Gamma, n{:}W \vdash \diamond}$$

(Exp n)
$$\frac{\Gamma', n{:}W, \Gamma'' \vdash \diamond}{\Gamma', n{:}W, \Gamma'' \vdash n : W}$$

(Env k)
$$\frac{\Gamma \vdash \diamond \quad k \notin dom(\Gamma)}{\Gamma, k{:}Bool \vdash \diamond}$$

(Proc Zero)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : T}$$

(Proc Par)
$$\frac{\Gamma \vdash P : T \quad \Gamma \vdash Q : T}{\Gamma \vdash P \mid Q : T}$$

(Proc Amb)
$$\frac{\Gamma \vdash M : \textbf{\textit{Amb}}[T] \quad \Gamma \vdash P : T}{\Gamma \vdash M[P] : S}$$

(Proc Repl)
$$\frac{\Gamma \vdash P : T}{\Gamma \vdash !P : T}$$

(Val True)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{true} : Bool}$$

(Proc Contex)
$$\frac{\Gamma, k : Bool \vdash M.P : \textbf{\textit{Cap}}[T]}{\Gamma \vdash k?M.P : \textbf{\textit{Cap}}[T]}$$

(Proc Ação)
$$\frac{\Gamma \vdash M : \textbf{\textit{Cap}}[T] \quad \Gamma \vdash P : \textbf{\textit{Cap}}[T]}{\Gamma \vdash M.P : \textbf{\textit{Cap}}[T]}$$

(Val False)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash (\neg\mathbf{true}) : Bool}$$

(Exp del)
$$\frac{\Gamma \vdash M[\mathbf{0}] : \textbf{\textit{Amb}}[T]}{\Gamma \vdash del\ M : \textbf{\textit{Cap}}[T]}$$

(Exp in)
$$\frac{\Gamma \vdash M : \textbf{\textit{Amb}}[S]}{\Gamma \vdash in\ M : \textbf{\textit{Cap}}[T]}$$

(Exp out)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash out : \textbf{\textit{Cap}}[T]}$$

(Proc Entrada ε)
$$\frac{\Gamma, n_1{:}\ W_1, \ \ldots\ , n_k{:}W_k \vdash P : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}{\Gamma \vdash (n_1{:}\ W_1, \ \ldots\ , n_k{:}W_k).P : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}$$

(Proc Saída ε)
$$\frac{\Gamma \vdash M_1 : W_1 \quad \ldots \quad \Gamma \vdash M_k : W_k}{\Gamma \vdash \langle M_1, \ \ldots\ , M_k \rangle : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}$$

(Proc Entrada α)
$$\frac{\Gamma, \tilde{n}{:}\ \tilde{W} \vdash P : \textbf{\textit{Cap}}[\tilde{W}\times] \qquad \Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}\times]] \qquad (\alpha \in \{\uparrow\downarrow ::\})}{\Gamma \vdash n\ \alpha(\tilde{n}{:}\ \tilde{W}).P : \textbf{\textit{Cap}}[\tilde{W}\times]}$$

(Exp Cont)
$$\frac{\Gamma \vdash P : T}{\Gamma \vdash C(P) : S}$$

(Proc Saída α)
$$\frac{\Gamma \vdash M_1 : W_1 \quad \ldots \quad \Gamma \vdash M_k : W_k \qquad \Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]] \qquad (\alpha \in \{\uparrow\downarrow ::\})}{\Gamma \vdash n\ \alpha\langle M_1, \ \ldots\ , M_k \rangle : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}$$

(Proc Abs ε)
$$\frac{\Gamma, n_1{:}\ W_1, \ \ldots\ , n_k{:}W_k \vdash x : \textbf{\textit{Abs}}[W_1 \times \ldots \times W_k] \qquad \Gamma \vdash P : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}{\Gamma \vdash x : \textbf{\textit{Abs}}[W_1 \times \ldots \times W_k] \triangleright (n_1{:}\ W_1, \ \ldots\ , n_k{:}W_k).P : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}$$

(Exp $\odot$)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \odot : T}$$

(Proc Abs α)
$$\frac{\Gamma, \tilde{n}{:}\ \tilde{W} \vdash x : \textbf{\textit{Abs}}[\tilde{W}\times] \qquad \Gamma \vdash P : \textbf{\textit{Cap}}[\tilde{W}\times] \qquad \Gamma \vdash m : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}\times]] \qquad (\alpha \in \{\uparrow\downarrow ::\})}{\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}\times] \triangleright m\ \alpha(\tilde{n}{:}\ \tilde{W}).P : \textbf{\textit{Cap}}[\tilde{W}\times]}$$

(Proc Cham ε)
$$\frac{\Gamma \vdash M_1 : W_1 \quad \ldots \quad \Gamma \vdash M_k : W_k \qquad \Gamma \vdash x : \textbf{\textit{Abs}}[W_1 \times \ldots \times W_k]}{\Gamma \vdash x\langle M_1, \ \ldots\ , M_k \rangle : \textbf{\textit{Cap}}[W_1 \times \ldots \times W_k]}$$

(Proc Res)
$$\frac{\Gamma, n : \textbf{\textit{Amb}}[T] \vdash P : S}{\Gamma \vdash (\nu n{:}\textbf{\textit{Amb}}[T])\ P : S}$$

(Proc Cham α)
$$\frac{\Gamma \vdash M_1 : W_1 \ldots \Gamma \vdash M_k : W_k \quad \Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}\times] \quad \Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}\times]] \quad (\alpha \in \{\uparrow\downarrow ::\})}{\Gamma \vdash n\ \alpha\ x\langle M_1, \ \ldots\ , M_k \rangle : \textbf{\textit{Cap}}[\tilde{W}\times]}$$

Figura 5.4 – Sistema de tipos para o CASC

(Proc Repl), (Exp in) e (Proc Res) são iguais às regras do sistema de tipos da Seção 4.2.1. A regra (Env k) é semelhante à regra (Env n) , porém é utilizada para adicionar uma variável do tipo $Bool$ em Γ. As regras (Proc True) e (Proc False) são básicas para o sistema de tipos, e servem para informar que **true** e **false** do tipo $Bool$ são valores válidos no sistema de tipos.

A regra (Proc Ação) é utilizada para tipar ações que não são precedidas por uma expressão de contexto, ou seja que não dependem de um contexto para serem executadas. Nesse caso, a capacidade M e o processo de continuidade P devem possuir o mesmo tipo para poder ser montada a expressão da ação. Em contrapartida, a regra (Proc Contex) é utilizada para tipar ações precedidas por uma expressão de contexto. Nesse caso, a expressão de contexto k é adicionada em Γ com o tipo $Bool$ e o restante da expressão é adicionada inteiramente em Γ, para posteriormente poder ser utilizada a regra (Proc Ação) para terminar de derivar a expressão.

A regra (Exp del) é utilizada para eliminar ambientes inativos, ou seja, que somente possuem processos **0**. Nesses ambientes, a capacidade del pode ser utilizada, e terá como tipo ***Cap***$[T]$, sendo T o mesmo tipo de troca do ambiente que será eliminado, ou seja, ***Amb***$[T]$. A regra (Exp out) é simples, visto que out não necessita de parâmetros. Nesse caso, a regra somente informa que out é uma capacidade bem tipada do tipo ***Cap***$[T]$ em Γ.

A regra (Exp Cont) é utilizada para tipar os contextos cuja sintaxe foi apresentada na Figura 4.10 da Seção 4.4.1. Um contexto C é um ambiente utilizado para modelar uma entidade que é relevante para uma aplicação. A expressão $C(P)$ corresponde à substituição do espaço reservado $\odot$ contido no contexto C pelo processo P, ou seja, $C\{\odot \leftarrow P\}$. Nesse caso, a regra convenciona que o tipo de troca da avaliação de contexto é S, ou seja, qualquer tipo e não necessariamente o mesmo tipo de P, pois uma avaliação de contexto não troca mensagens, somente o processo que ele representa. Por exemplo, considere o contexto C modelado abaixo:

$$C \triangleq conf[P \mid bob[\odot] \mid alice[Q]] : T \qquad \text{(Contexto } C\text{)}$$

A expressão representada pelo contexto C possui o tipo de troca T. Porém a avaliação do contexto não necessariamente possuirá o mesmo T, por exemplo, $C(cel[R]) : S$. Essa regra é fundamental para provar a "relação de redução de processos", que será demonstrada na Seção 5.3. A regra (Exp $\odot$) convenciona que o espaço reservado $\odot$ pode ter qualquer tipo T. Dessa forma, $\odot$ pode ser colocado em paralelo com qualquer outro processo, como por exemplo, $bob[\odot \mid P : T]$ (mesma lógica da regra (Proc Zero)).

Com relação à troca de mensagens, a regra (Proc Entrada ε) convenciona que, a qualquer momento, pode ser adicionada uma ou mais variáveis do tipo W em Γ e, caso exista uma

capacidade com o mesmo tipo de troca das novas variáveis (***Cap***$[W]$), é possível montar uma expressão para entrada de dados. A regra (Proc Saída ε) convenciona que se existe uma ou mais mensagens do tipo W em Γ, o envio dessa mensagem no ambiente será uma capacidade com mesmo tipo de troca W, ou seja, ***Cap***$[W]$. A regra (Proc Entrada α) é similar à regra (Proc Entrada ε), porém, na regra α é possível informar de qual ambiente se espera a mensagem, representado por n. Obrigatoriamente o ambiente n deve possuir tipo de troca da expressão de entrada, ou seja ***Amb***$[$***Cap***$[W]]$. Além disso, deve-se informar a localização do ambiente, representado pelo identificador α. A localização pode ser "em algum nível superior" ($\uparrow$), "em algum nível inferior" ($\downarrow$) ou "no mesmo nível" ($::$). A regra (Proc Saída α) segue a mesma lógica.

As últimas quatro regras são utilizadas para as abstrações de processo. A regra (Proc Abs ε) convenciona que a qualquer momento pode ser adicionada uma ou mais variáveis do tipo W em Γ e, caso exista uma variável do tipo abstração de processo com o mesmo tipo de troca W, (ou seja, ***Abs***$[W]$) e uma capacidade com o mesmo tipo de troca das novas variáveis (***Cap***$[W]$), é possível criar uma expressão para vincular a variável x à capacidade P, que possui como parâmetros de entrada as variáveis W adicionadas em Γ. A regra (Proc Abs α) permite informar o ambiente onde está o processo que será vinculado a x. Ela segue a mesma lógica da regra (Proc Entrada α). A regra (Proc Cham ε) convenciona que se existem uma ou mais mensagens do tipo W em Γ, e uma abstração de processo com tipo de troca W, ou seja, ***Abs***$[W]$, é possível montar uma expressão para invocar a abstração de processo, enviando como parâmetro as mensagens do tipo W. A invocação da abstração de processo será uma capacidade com mesmo tipo de troca W, ou seja, ***Cap***$[W]$. Por fim, a regra (Proc Cham α) segue a mesma ideia da regra (Proc Cham ε), porém permite informar o ambiente onde está localizada a abstração de processo x que está sendo invocada.

Com base nas regras propostas é possível derivar expressão "bob phone"(adaptado de (SIEWE; ZEDAN; CAU, 2011)):

$$bob[phone[at(conf)?switchto\langle silent\rangle.\mathbf{0} \mid out.\mathbf{0}]] : Shh \qquad \text{(bob phone)}$$

Para a expressão "bob phone" ser bem tipada, ou seja, poder ser derivada corretamente, deve-se conhecer previamente os tipos dos ambientes (para o *typechecker* saber quais tipos os processos rodando dentro do ambiente devem possuir), das mensagens a serem enviadas (para o *typechecker* confirmar se o tipo da mensagem corresponde ao tipo permitido no ambiente) e da abstração de processo, ou seja: bob: ***Amb***$[Shh]$, $phone$: ***Amb***$[$***Cap***$[W]]$, $silent$: W,

$switchto$: ***Abs***$[W]$. A árvore de derivação dessa expressão que prova que ela é bem tipada pode ser visualizada na Figura A.4 do Apêndice A. A mesma prova não pode ser concluída se $phone$: ***Amb***$[Shh]$, o que está correto, pois $phone$ não pode ser Shh já que existem processos trocando mensagens dentro dele.

Outro exemplo que pode ser tipado é o exemplo do "projetor" apresentado no início desta Seção.

$$sala[proj[pess :: (x : W).P] \mid pess[proj :: \langle M \rangle]] : Shh \qquad \text{(projetor)}$$

Esse exemplo demonstra a comunicação entre ambientes localizados no mesmo nível de uma hierarquia (irmãos). Nesse caso, o sistema de tipos deve permitir a comunicação somente se os tipos dos ambientes combinarem. Para esse exemplo ser bem tipado, deve-se conhecer previamente que $sala$: ***Amb***$[Shh]$, $proj$, $pess$: ***Amb***[***Cap***$[W]$], P: W e M: W. A árvore de derivação dessa expressão pode ser visualizada na Figura A.5 do Apêndice A.

Congruência Estrutural

$P \equiv P$	(Estrut Refl)
$P \equiv Q \quad \Rightarrow \quad Q \equiv P$	(Estrut Simm)
$P \equiv Q,\ Q \equiv R \quad \Rightarrow \quad P \equiv R$	(Estrut Trans)
$P \equiv Q \quad \Rightarrow \quad (\nu n\colon S)P \equiv (\nu n\colon S)Q$	(Estrut Res)
$P \equiv Q \quad \Rightarrow \quad P \mid R \equiv Q \mid R$	(Estrut Par)
$P \equiv Q \quad \Rightarrow \quad !P \equiv !Q$	(Estrut Repl)
$P \equiv Q \quad \Rightarrow \quad n[P] \equiv n[Q]$	(Estrut Amb)
$P \equiv Q \quad \Rightarrow \quad M.P \equiv M.Q$	(Estrut Ação)
$P \mid Q \equiv Q \mid P$	(Estrut Par Com)
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	(Estrut Par Ass)
$(\nu n\colon S)(\nu m\colon U)P \equiv (\nu m\colon U)(\nu n\colon S)P \qquad se\ n \neq m$	(Estrut Res Res)
$(\nu n\colon S)(P \mid Q) \equiv P \mid (\nu n\colon S)Q \qquad se\ n \notin fn(P)$	(Estrut Res Par)
$(\nu n\colon S)m[P] \equiv m[(\nu n\colon S)P] \qquad se\ n \neq m$	(Estrut Res Amb)
$!P \equiv P \mid !P$	(Estrut Repl Par)
$P \mid \mathbf{0} \equiv P$	(Estrut Zero Par)
$!\mathbf{0} \equiv \mathbf{0}$	(Estrut Zero Repl)
$\mathbf{0}.P \equiv P$	(Estrut Zero Ação)
$(\nu n\colon S)\mathbf{0} \equiv \mathbf{0}$	(Estrut Zero Res)
$\mathbf{true}?M.P \equiv M.P$	(Estrut Ação)
$P \equiv Q, (\models k \Leftrightarrow k') \quad \Rightarrow \quad k?P \equiv k'?Q$	(Estrut Cont Ação)
$P \equiv Q \quad \Rightarrow \quad x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P \equiv x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).Q$	(Estrut Abs Proc)
$del\ n.P \mid n[\mathbf{0}] \equiv P$	(Estrut Del)

Figura 5.5 – Congruência Estrutural do CASC

A congruência estrutural do CASC precisou ser alterada em função do sistema de tipos (ficando diferente da formalizada em (SIEWE; ZEDAN; CAU, 2011)). Nas congruências

(Estrut Res), (Estrut Res Res), (Estrut Res Par), (Estrut Res Amb), (Estrut Zero Res) e (Estrut Abs Proc) foram explicitamente adicionados tipos nas expressões. Como a congruência será necessária para provar que o sistema de tipos apresenta a propriedade de *preservação*, a mesma pode ser visualizada na Figura 5.5.

Em função da nova sintaxe, as relações de reduções dos processos foram alteradas. Foram removidas várias reduções que permitiam a comunicação entre qualquer ambiente pai, filho ou irmão (conforme explicado no início do capítulo, a sintaxe do CASC foi alterada para somente permitir a comunicação entre ambiente através do nome), além de adicionados explicitamente tipos para algumas reduções. As novas reduções podem ser visualizadas na Figura 5.6.

Relação de redução de processos

$x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P \mid x\langle\tilde{z}\rangle \quad \rightarrow \quad x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P \mid P\{\tilde{y} \leftarrow \tilde{z}\}$	(Red Call Lc)
$n[Q \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P] \mid m[n :: x\langle\tilde{z}\rangle \mid R] \quad \rightarrow$ $n[Q \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P] \mid m[P\{\tilde{y} \leftarrow \tilde{z}\} \mid R]$	(Red Call S)
$n[Q \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P \mid m[n \uparrow x\langle\tilde{z}\rangle \mid R]] \quad \rightarrow$ $n[Q \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P \mid m[P\{\tilde{y} \leftarrow \tilde{z}\} \mid R]]$	(Red Call U)
$Q \mid m \downarrow x\langle\tilde{z}\rangle \mid m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P] \quad \rightarrow$ $Q \mid P\{\tilde{y} \leftarrow \tilde{z}\} \mid m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P]$	(Red Call D)
$(\tilde{y} : \tilde{W}).P \mid \langle\tilde{z}\rangle.Q \quad \rightarrow \quad P\{\tilde{y} \leftarrow \tilde{z}\} \mid Q$	(Red Com Lc)
$n[m :: (\tilde{y} : \tilde{W}).P.\mathbf{0} \mid Q] \mid m[n :: \langle\tilde{z}\rangle.R \mid S] \quad \rightarrow \quad n[P\{\tilde{y} \leftarrow \tilde{z}\} \mid Q] \mid m[R \mid S]$	(Red Com S)
$u[n \downarrow (\tilde{y} : \tilde{W}).P.\mathbf{0} \mid n[u \uparrow \langle\tilde{z}\rangle.Q \mid R] \mid S] \quad \rightarrow \quad u[P\{\tilde{y} \leftarrow \tilde{z}\} \mid n[Q \mid R] \mid S]$	(Red Com R1)
$u[n \downarrow \langle\tilde{z}\rangle.P \mid n[[u \uparrow (\tilde{y} : \tilde{W}).Q \mid R] \mid S] \quad \rightarrow \quad u[P \mid n[Q\{\tilde{y} \leftarrow \tilde{z}\} \mid R] \mid S]$	(Red Com R2)
$n[in\ m.P \mid Q] \mid m[R] \quad \rightarrow \quad m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out.P \mid Q] \mid R] \quad \rightarrow \quad n[P \mid Q] \mid m[R]$	(Red Out)
$P \rightarrow Q \quad \Rightarrow \quad (\nu n : S)P \rightarrow (\nu n : S)Q$	(Red Res)
$P \rightarrow Q \quad \Rightarrow \quad n[P] \rightarrow n[Q]$	(Red Amb)
$P \rightarrow Q \quad \Rightarrow \quad P \mid R \rightarrow Q \mid R$	(Red Par)
$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \quad \Rightarrow \quad P' \rightarrow Q'$	(Red $\equiv$)
$P \rightarrow Q \quad \Rightarrow \quad C(P) \rightarrow C(Q)$	(Red Cont)
$C(M.P) \rightarrow C'(P) \quad \Rightarrow \quad C(k?M.P) \rightarrow C'(P) \qquad$ se $C \models k$	(Red Guard)

Figura 5.6 – Relação de redução de processos no CASC

5.2 Estudos de caso

Para exemplificar o uso do CASC juntamente com o sistema de tipos proposto, serão modelados dois cenários.

O primeiro cenário (AUGUSTO, 2007) baseia-se em um hospital. Considere um quarto de hospital onde um paciente encontra-se internado e sendo monitorado por razões de saúde e de

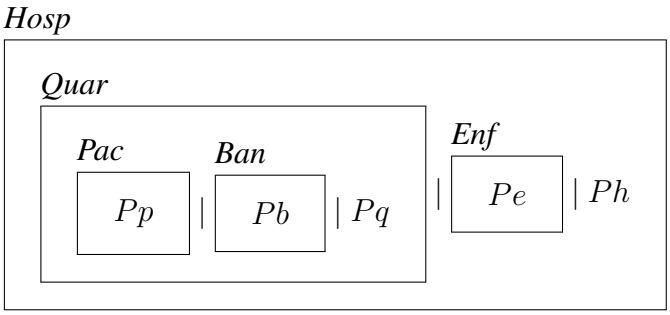
segurança. Dentro do quarto encontram-se móveis, equipamentos médicos, um banheiro e uma janela. Sensores e pulseiras podem ser utilizadas para monitorar quem entra ou sai do quarto ou se aproxima de áreas específicas, tais como janela, porta ou banheiro. Um dos contextos de interesse pode ser enviar uma mensagem para uma enfermeira quando um paciente estiver há mais de 20 minutos no banheiro ou se um paciente em estado delicado sair do seu quarto.

Lendo o cenário é possível modelar 5 ambientes e seus respectivos processos, conforme Tabela 5.1.

Tabela 5.1 – Ambientes e processos do cenário 1

Entidade	**Nome do Ambiente**	**Processos Internos**
Hospital	Hosp	Ph
Quarto	Quar	Pq
Banheiro	Ban	Pb
Paciente	Pac	Pp
Enfermeira	Enf	Pe

Modelando o cenário na forma de ambientes:



O primeiro passo é detalhar o processo Pp (Pulseira do Paciente). Ela deverá enviar uma mensagem de alerta para uma enfermeira caso ocorra alguma situação de perigo (saiu do quarto, há mais de 20 minutos no banheiro, etc). Este controle pode ser modelado através de uma expressão de contexto chamada $Perigo$. A modelagem dessa expressão não será detalhada nesse exemplo, somente é necessário saber que ela retornará **true**, caso alguma situação de perigo prevista aconteça. Se a enfermeira já estiver dentro do quarto, ela não precisará ser notificada (pois estará presenciando a situação de perigo), somente se ela estiver fora do quarto. Como o ambiente enfermeira não é o ambiente superior do paciente, e nem do quarto, não é possível utilizar a localização superior ↑, nem ambiente irmão :: para enviar a mensagem. Uma alternativa é modelar o processo do hospital (Ph) para ser um propagador de mensagens. Nesse caso, a pulseira do paciente enviará a mensagem diretamente ao ambiente hospital e ele, por sua vez, enviará a mensagem para a enfermeira, caso ela esteja localizada no mesmo nível. Caso

contrário, entende-se que ela já esteja dentro do quarto.

a) $Pp \triangleq Perigo?Hosp \uparrow EnviaMsg\langle AL\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W]$

b) $Ph \triangleq EnviaMsg : \textbf{\textit{Abs}}[W] \triangleright Pac \downarrow (M : W).Enf \downarrow \langle M\rangle : \textbf{\textit{Cap}}[W]$

c) $Pe \triangleq Hosp \uparrow (M : W).\mathbf{0} : \textbf{\textit{Cap}}[W]$ (Expressões 1)

A expressão **a** modela a pulseira do paciente que será responsável por enviar a mensagem para o ambiente hospital. A expressão de contexto $Perigo$ é responsável por monitorar o paciente quando ele está há mais de 20 minutos no banheiro ou quando sair do quarto (caso seja um paciente em estado delicado), retornando **true** quando a mensagem precisar ser enviada. A mensagem é enviada quando a abstração de processo $EnviaMsg$ é invocada passando a mensagem de alerta como parâmetro. O processo responsável por replicar a mensagem até a enfermeira foi transformado em uma abstração de processo, conforme pode ser visualizado na expressão **b**. A abstração espera uma mensagem de um ambiente paciente localizado em algum ambiente filho, e ao receber a mensagem, imediatamente a retransmite a um ambiente filho enfermeira. O processo rodando no ambiente enfermeira (Pe) por sua vez, espera uma mensagem do ambiente superior hospital (expressão **c**).

Nas expressões **b** e **c**, o sistema de tipos já está sendo utilizado, tanto para colocar tipos na abstração de processo, quanto para tipar as mensagens enviadas e recebidas. Dessa forma, não é possível enviar uma mensagem diferente (com mais um de um parâmetro ou com um tipo diferente) para o hospital ou para a pulseira da enfermeira. As provas dessas expressões podem ser visualizadas na Figura A.6 do Apêndice A.

O processo rodando dentro do quarto do paciente (Pq) poder ser um sensor responsável por guardar informações sobre o paciente, tais como dados pessoais, tipo de tratamento, médico e enfermeira responsáveis, etc. Além de liberar o acesso à essas informações, quando requisitado. Esse processo espera uma mensagem de um ambiente enfermeira, que esteja dentro do quarto, ou seja, um ambiente filho. Além disso, somente enfermeiras autorizadas podem solicitar informações do paciente. Não é possível acessar essa informação de outro ambiente que não seja enfermeira, e que não esteja dentro do quarto. A expressão Pq pode ser modelada da seguinte forma:

$$Pq \triangleq Enf \downarrow (Login : W).P : \textbf{\textit{Cap}}[W] \quad \text{(Expressão Pq)}$$

Essa expressão espera receber um parâmetro como mensagem: o $Login$ que consiste nas informações utilizadas pela enfermeira para autenticar-se (segurança do sistema). O processo P

pode ser uma função que, após autenticar-se, retornaria todas as informações do paciente, que fossem úteis para a enfermeira. Essa expressão não será derivada, pois somente uma regra de tipo é necessária para provar que está correta: (Proc Entrada α).

Por fim, falta modelar a expressão que será utilizada pela enfermeira para entrar no quarto e solicitar as informações do paciente. Essa expressão estará rodando em paralelo com a expressão **c** modelada anteriormente.

$$Pe \triangleq Hosp \uparrow (M : W).\mathbf{0} : \textbf{\textit{Cap}}[W] \mid$$
$$in\ Quar.Quar \uparrow \langle Aut \rangle : \textbf{\textit{Cap}}[W] \qquad \text{(Expressão Pe)}$$

Se a enfermeira conseguir entrar no quarto, ou seja, se a expressão $in\ Quar$ for executada, então uma mensagem é enviada ao ambiente passando as informações de autenticação da enfermeira ($Login$). Após a execução do comando in, o ambiente enfermeira está dentro do ambiente quarto, ou seja, tornando-se filho de quarto, e fazendo com que a expressão Pq receba a mensagem. A prova dessa expressão pode ser visualizada na Figura A.7 do Apêndice A. Conforme pode ser visualizado nas árvores de derivação, o sistema de tipos exige que os tipos dos ambientes envolvidos ($Hosp$, Pac, Enf) nas trocas de mensagens sejam iguais. Isso é necessário para o sistema de tipos garantir que a comunicação será feita de forma correta. Além disso, neste exemplo os ambientes trocam mensagens diretamente entre si, por essa razão se justifica terem o mesmo tipo. Um alternativa para ambientes que tenham tipos diferentes e que necessitam trocar mensagens é criar um ambiente novo (através da regra de tipo Proc Res) no ambiente remetente com o tipo do ambiente de destino da mensagem e utilizá-lo somente para entregar a mensagem. Nesse caso, o ambiente auxiliar utilizaria a capacidade out para sair do ambiente remetente, e in para entrar no ambiente de destino, e assim enviar a mensagem.

O segundo cenário a ser modelado baseia-se no artigo de Lee (2009). Em um dia muito quente, Pedro está voltando para sua casa de ônibus. Quando faltam alguns minutos para chegar em casa o seu celular mostra uma mensagem: “Hoje é um dia muito quente, gostaria de beber um *cappuccino* gelado quando chegar em casa?”. Pedro responde “sim” e ao chegar em casa a máquina de *cappuccino* já preparou a bebida. Esse cenário demonstra a comunicação entre dispositivos heterogêneos que as aplicações pervasivas necessitam.

Lendo o cenário é possível modelar seis ambientes e seus respectivos processos, conforme Tabela 5.2.

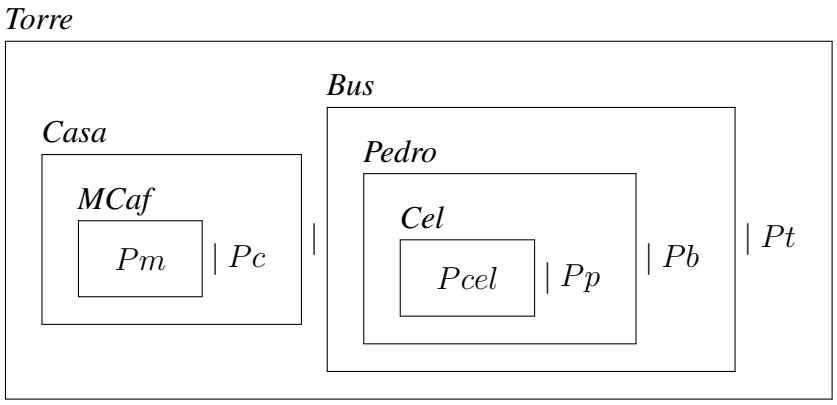
O ambiente “Torre de Comunicação” será o ambiente raiz, ou seja, todos os outros estarão contidos dentro dele. Ele representará o alcance (ou a abrangência) do sinal do celular,

Tabela 5.2 – Ambientes e processos do cenário 2

Entidade	Nome do Ambiente	Processos Internos
Ônibus	Bus	Pb
Casa do Pedro	Casa	Pc
Máquina de Café	MCaf	Pm
Pedro	Pedro	Pp
Celular do Pedro	Cel	Pcel
Torre de Comunicação	Torre	Pt

visto que ele será utilizado para se comunicar com a máquina de café de Pedro.

Modelando o cenário na forma de ambientes, temos:



O processo Pcel será responsável por monitorar a temperatura do dia e mostrar a mensagem para Pedro. O monitoramento da temperatura pode ser feito através de uma expressão de contexto *DiaQuente*. Essa expressão pode usar informações metereológicas obtidas através da *Internet* pelo próprio celular. Quando a expressão retornar **true**, uma mensagem é enviada para o celular. Rodando em paralelo, deverá existir uma expressão responsável por enviar a mensagem à torre do celular. A torre, por sua vez, retransmite a mensagem para a máquina de café. Esse comportamento será modelado através de uma abstração de processo chamada *EnviaMsg* que receberá como parâmetro a mensagem a ser retransmitida. Por fim, a máquina de café espera receber uma mensagem da torre. Essas expressões podem ser modeladas da seguinte forma:

a) $Pcel \triangleq DiaQuente?MostraMsg\langle Pergunta\rangle : \textbf{\textit{Cap}}[W] \mid EnviaMsg\langle M\rangle : \textbf{\textit{Cap}}[W]$

b) $Pt \triangleq EnviaMsg : \textbf{\textit{Abs}}[W] \triangleright Cel \downarrow (M : W).MCaf \downarrow \langle M\rangle : \textbf{\textit{Cap}}[W]$

c) $Pm \triangleq Torre \uparrow (M : W).\mathbf{0} : \textbf{\textit{Cap}}[W]$ (Expressões 2)

As árvores de derivação dessas expressões são semelhantes às do cenário 1, portanto

não serão demonstradas. Uma melhoria que seria necessária ao sistema de tipos seria facilitar o envio de nomes de ambientes como parâmetros. Por exemplo, a abstração de processo da expressão **b)**, além de receber a mensagem poderia receber por parâmetro o nome de ambiente de destino, ao invés de ficar fixo *MCaf*. Nesse caso, a expressão ficaria da seguinte forma:

$$Pcel \triangleq EnviaMsg : \textbf{\textit{Abs}}[W_1 \times W_2] \triangleright Cel \downarrow (M : W_1).$$
$$\underbrace{N : W_2 \downarrow \langle M \rangle : \textbf{\textit{Cap}}[W_1 \times W_2]}_{\text{regra de tipo (Proc Saída } \alpha)}$$

Conforme a regra de tipo (Proc Saída α), $N = \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W_1 \times W_2]]$, portanto $W_2 = \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W_1 \times W_2]]$, ou seja, um tanto quanto confusa e recursiva a representação. Esse problema será abordado na seção de trabalhos futuros.

5.3 Provas

O sistema de tipos atribui para cada termo no máximo um tipo, e existe no máximo uma árvore de derivação para a prova do tipo do termo. As propriedades da relação de tipos são em geral provadas por *indução* nas árvores de derivação (DERANSART; SMAUS, 2002).

Uma das principais propriedades de um sistema de tipos é chamada *preservation* ou *subject reduction*, conforme explicado na Seção 3. Ela garante que se um termo bem tipado pode ser avaliado, então o termo resultando da avaliação também é bem tipado. A outra propriedade de um sistema de tipos é a *progress*, que garante que um termo ou é um valor, ou seja, não pode mais ser avaliado, ou então ele pode ser avaliado gerando um novo termo (conforme as regras de reduções). Nesta dissertação, somente será provada a propriedade *subject reduction*, pois para provar *progress* é necessário definir os valores finais do sistema de tipos. Esta questão é abordada em trabalhos futuros, no Capítulo 6.

Como os termos possuem uma relação de equivalência (congruência) também deve-se provar *subject reduction* para a congruência (Figura 5.5). Além da relação de redução entre os processos (Figura 5.6). Para algumas provas são necessários lemas auxiliares. Considere X como sendo uma sentença qualquer.

5.3.1 Lema (Sentença Implícita)

Se $\Gamma', \Gamma'' \vdash X$ *então* $\Gamma' \vdash \diamond$

5.3.2 Lema (Troca)

Se $\Gamma', n : W', m : W'', \Gamma'' \vdash X$ *então* $\Gamma', m : W'', n : W', \Gamma'' \vdash X$.

5.3.3 Lema

Se $\Gamma', \Gamma'' \vdash X$ *e* $n \notin dom(\Gamma', \Gamma'')$ *então* $\Gamma', n : W, \Gamma'' \vdash X$.

5.3.4 Lema

Se $\Gamma', \Gamma'' \vdash X$ *e* $t \notin dom(\Gamma', \Gamma'')$ *então* $\Gamma', t : Bool, \Gamma'' \vdash X$.

5.3.5 Lema

Se $\Gamma', n : W', \Gamma'' \vdash X$ *e* $n \notin fn(X)$ *então* $\Gamma', \Gamma'' \vdash X$.

5.3.6 Lema (Substituição)

Se $\Gamma', \tilde{y} : \tilde{W}, \Gamma'' \vdash X$ *e* $\Gamma' \vdash \tilde{z} : \tilde{W}$ *então* $\Gamma', \Gamma'' \vdash X\{\tilde{y} \leftarrow \tilde{z}\}$.

5.3.7 Teorema (Subject Congruence)

(1) *Se* $\Gamma \vdash P : T$ *e* $P \equiv Q$ *então* $\Gamma \vdash Q : T$.

(2) *Se* $\Gamma \vdash P : T$ *e* $Q \equiv P$ *então* $\Gamma \vdash Q : T$.

Prova. A prova é por indução mutual nas derivações $P \equiv Q$ e $Q \equiv P$.

(1) Se $\Gamma \vdash P : T$ e $P \equiv Q$ então $\Gamma \vdash Q : T$

(Estrut Refl) Então $P \equiv P$. Trivial.

(Estrut Simm) Assumimos por **(1)** que $\Gamma \vdash P : T$ e $P \equiv Q$. Pela regra da simetria, temos então que $Q \equiv P$. E pela hipótese de **(2)**, temos que $\Gamma \vdash Q : T$.

(Estrut Trans) Então $P \equiv R$ e $R \equiv Q$ para algum R. Pela hipótese da indução **(1)**, é possível concluir que $\Gamma \vdash R : T$. Aplicando novamente a hipótese da indução, temos que $\Gamma \vdash Q : T$.

(Estrut Res) Então $(\nu n\colon S)P \equiv (\nu n\colon S)Q$. É possível assumir por **(1)** que $\Gamma \vdash (\nu n\colon S)P : T$, sendo que $S = \textbf{\textit{Amb}}[T]$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Res), com a premissa $\Gamma, n\colon S \vdash P : T$ e pela hipótese indutiva têm-se que $\Gamma, n\colon S \vdash Q : T$. Aplicando (Proc Res) nessa premissa conclui-se que $\Gamma \vdash (\nu n\colon S)Q : T$.

(Estrut Par) Então $P \mid R \equiv Q \mid R$. É possível assumir por **(1)** que $\Gamma \vdash P \mid R : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash P : T$ e $\Gamma \vdash R : T$. Pela hipótese da indução ($P \equiv Q$) e por **(1)** é possível concluir que $\Gamma \vdash Q : T$. Aplicando (Proc Par) nas premissas $\Gamma \vdash Q : T$ e $\Gamma \vdash R : T$, conclui-se que $\Gamma \vdash Q \mid R : T$.

(Estrut Repl) Então $!P \equiv !Q$. É possível assumir por **(1)** que $\Gamma \vdash !P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Repl) com a premissa $\Gamma \vdash P : T$. Pela hipótese da indução ($P \equiv Q$) e por **(1)**, é possível concluir que $\Gamma \vdash Q : T$. Aplicando (Proc Repl) nessa premissa, conclui-se que $\Gamma \vdash !Q : T$.

(Estrut Amb) Então $n[P] \equiv n[Q]$. É possível assumir por **(1)** que $\Gamma \vdash n[P] : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Amb) com a premissa $\Gamma \vdash n : \textbf{\textit{Amb}}[U]$ e $\Gamma \vdash P : U$. Pela hipótese da indução ($P \equiv Q$) e por **(1)**, é possível concluir que $\Gamma \vdash Q : U$. Aplicando (Proc Amb) nas premissas $\Gamma \vdash n : \textbf{\textit{Amb}}[U]$ e $\Gamma \vdash Q : U$, conclui-se que $\Gamma \vdash n[Q] : T$.

(Estrut Ação) Então $M.P \equiv M.Q$. É possível assumir por **(1)** que $\Gamma \vdash M.P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Ação) com as premissas $\Gamma \vdash M : T$ e $\Gamma \vdash P : T$, sendo que $T = \textbf{\textit{Cap}}[U]$. Pela hipótese da indução ($P \equiv Q$) e por **(1)**, é possível concluir que $\Gamma \vdash Q : T$. Aplicando (Proc Ação) nas premissas $\Gamma \vdash M : T$ e $\Gamma \vdash Q : T$, conclui-se que $\Gamma \vdash M.Q : T$.

(Estrut Par Com) Então $P \mid Q \equiv Q \mid P$. É possível assumir por **(1)** que $\Gamma \vdash P \mid Q : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash P : T$ e $\Gamma \vdash Q : T$. Aplicando novamente (Proc Par) nessas premissas, é possível concluir que $\Gamma \vdash Q \mid P : T$.

(Estrut Par Ass) Então $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$. É possível assumir por **(1)** que $\Gamma \vdash (P \mid Q) \mid R : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash (P \mid Q) : T$ e $\Gamma \vdash R : T$. A primeira premissa também deve ter derivado de (Proc Par) com $\Gamma \vdash P : T$ e $\Gamma \vdash Q : T$. Aplicando (Proc Par) duas vezes nas premissas, é possível concluir que $\Gamma \vdash P \mid (Q \mid R)$.

(Estrut Res Res) Então $(\nu n\colon S)(\nu m\colon U)P \equiv (\nu m\colon U)(\nu n\colon S)P$, se $n \neq m$. É possível assumir por **(1)** que $\Gamma \vdash (\nu n\colon S)(\nu m\colon U)P : T$. Essa conclusão deve ter sido derivada duas vezes da regra de tipo (Proc Res) com as premissas $\Gamma, n : \textbf{\textit{Amb}}[V], m : \textbf{\textit{Amb}}[Z] \vdash P : T$, sendo que $S = \textbf{\textit{Amb}}[V]$ e $U = \textbf{\textit{Amb}}[Z]$. Aplicando o lema 5.3.2, conclui-se que

$\Gamma, m : \textbf{\textit{Amb}}[Z], n : \textbf{\textit{Amb}}[V] \vdash P : T$. Aplicando (Proc Res) duas vezes nessa premissa, conclui-se que $\Gamma \vdash (\nu m\colon U)(\nu n\colon S)P : T$.

(Estrut Res Par) Então $(\nu n\colon S)(P \mid Q) \equiv P \mid (\nu n\colon S)Q$ se $n \notin fn(P)$. É possível assumir por **(1)** que $\Gamma \vdash (\nu n\colon S)(P \mid Q) : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Res) com a premissa **(a)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash P \mid Q : T$, sendo que $S = \textbf{\textit{Amb}}[V]$. Essa premissa deriva de (Proc Par) com **(b)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash P : T$ e **(c)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash Q : T$. Pelo lema 5.3.5, desde que $n \notin fn(P)$ conclui-se que **(d)** $\Gamma \vdash P : T$. Aplicando (Proc Res) na premissa **(c)**, conclui-se que $\Gamma \vdash (\nu n : \textbf{\textit{Amb}}[V])Q : T$ e aplicando (Proc Par) nessa conclusão mais a premissa **(d)**, conclui-se $\Gamma \vdash P \mid (\nu n : \textbf{\textit{Amb}}[V])Q : T$.

(Estrut Res Amb) Então $(\nu n\colon S)m[P] \equiv m[(\nu n\colon S)P]$ se $n \neq m$. É possível assumir por **(1)** que $\Gamma \vdash (\nu n\colon S)m[P] : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Res) com **(a)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash m[P] : T$, sendo que $S = \textbf{\textit{Amb}}[V]$. A premissa **(a)** deriva de (Proc Amb) com **(b)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash m : \textbf{\textit{Amb}}[U]$ e **(c)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash P : U$ para algum U. Pelo lema 5.3.5, desde que $n \neq m$, é possível concluir que **(d)** $\Gamma \vdash m : \textbf{\textit{Amb}}[U]$ Aplicando (Proc Res) em **(c)**, conclui-se $\Gamma \vdash (\nu n : \textbf{\textit{Amb}}[V])P : U$. Aplicando (Proc Amb) nessa conclusão mais a premissa **(d)**, conclui-se $\Gamma \vdash m[(\nu n : \textbf{\textit{Amb}}[V])P] : T$.

(Estrut Repl Par) Então $!P \equiv P \mid !P$. É possível assumir por **(1)** que $\Gamma \vdash !P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Repl) com a premissa $\Gamma \vdash P : T$. Aplicando (Proc Par) conclui-se $P \mid !P : T$.

(Estrut Zero Par) Então $P \mid \mathbf{0} \equiv P$. É possível assumir por **(1)** que $\Gamma \vdash P \mid \mathbf{0} : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash P : T$ e $\Gamma \vdash \mathbf{0} : T$.

(Estrut Zero Repl) Então $!\mathbf{0} \equiv \mathbf{0}$. É possível assumir por **(1)** que $\Gamma \vdash !\mathbf{0} : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Repl) com $\mathbf{0} : T$.

(Estrut Zero Ação) Então $\mathbf{0}.P \equiv P$. É possível assumir por **(1)** que $\Gamma \vdash \mathbf{0}.P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Ação) com $\Gamma \vdash \mathbf{0} : T$ e $\Gamma \vdash P : T$, sendo que $T = \textbf{\textit{Cap}}[S]$ para algum S.

(Estrut Zero Res) Então $(\nu n\colon S)\mathbf{0} \equiv \mathbf{0}$. É possível assumir por **(1)** que $\Gamma \vdash (\nu n\colon S)\mathbf{0} : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Res) com a premissa $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash \mathbf{0} : T$, sendo que $S = \textbf{\textit{Amb}}[V]$. Pelo lema 5.3.5 é possível concluir que $\Gamma \vdash \mathbf{0} : T$.

(Estrut Ação) Então $\textbf{true}?M.P \equiv M.P$. É possível assumir por **(1)** que $\Gamma \vdash \textbf{true}?M.P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Contex) com a premissa $\Gamma, \textbf{true} : Bool \vdash M.P : T$, sendo que $T = \textbf{\textit{Cap}}[S]$.

(Estrut Cont Ação) Então $k?P \equiv k'?Q$. É possível assumir por **(1)** que $\Gamma \vdash k?P : T$. Pela hipótese da indução têm-se que $P \equiv Q$ e $k \Leftrightarrow k'$. Assim é possível concluir por **(1)** que $\Gamma \vdash k'?Q : T$.

(Estrut Abs Proc) Então $x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P \equiv x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).Q$. É possível assumir por **(1)** que $x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Abs ε) com as premissas $\Gamma, n : \tilde{y} : \tilde{W} \vdash x : \textbf{\textit{Abs}}[\tilde{W}]$ e $\Gamma \vdash P : T$, sendo que $T = \textbf{\textit{Cap}}[\tilde{W}]$. Pela hipótese da indução ($P \equiv Q$) e por **(1)** é possível concluir que $\Gamma, n : \tilde{y} : \tilde{W} \vdash x : \textbf{\textit{Abs}}[\tilde{W}]$ e $\Gamma \vdash Q : T$. Aplicando (Proc Abs ε) nessas premissas conclui-se que $x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).Q : T$.

(Estrut Del) Então $del\ n.P \mid n[\mathbf{0}] \equiv P$. Considere $P = del\ n.P' \mid n[\mathbf{0}]$ e $Q = P'$. É possível assumir por **(1)** que $\Gamma \vdash P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash del\ n.P' : T$ e **(b)** $\Gamma \vdash n[\mathbf{0}] : T$. A conclusão **(a)** deve ter sido derivada da regra de tipo (Proc Ação) com as premissas $\Gamma \vdash del\ n : \textbf{\textit{Cap}}[S]$ e $\Gamma \vdash P' : \textbf{\textit{Cap}}[S]$, sendo que $T = \textbf{\textit{Cap}}[S]$. Como $Q = P'$ é possível assumir por **(1)** que $\Gamma \vdash Q : T$.

(2) Se $\Gamma \vdash P : T$ e $Q \equiv P$ então $\Gamma \vdash Q : T$

(Estrut Refl) Então $P \equiv P$. Trivial.

(Estrut Simm) Assumimos por **(2)** que $\Gamma \vdash P : T$ e $Q \equiv P$. Pela regra da simetria temos então que $P \equiv Q$. E pela hipótese de **(1)**, temos que $\Gamma \vdash Q : T$.

(Estrut Trans) Então $Q \equiv R$ e $R \equiv P$, para algum R. Pela hipótese da indução **(2)** é possível concluir que $\Gamma \vdash R : T$ e $\Gamma \vdash Q : T$.

(Estrut Res), (Estrut Par), (Estrut Repl), (Estrut Amb), (Estrut Ação), (Estrut Par Ass) Simétrico ao caso **(1)**.

(Estrut Par Com) Então $P \mid Q \equiv Q \mid P$. É possível assumir por **(2)** que $\Gamma \vdash Q \mid P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash Q : T$ e $\Gamma \vdash P : T$. Aplicando novamente (Proc Par) nessas premissas é possível concluir que $\Gamma \vdash P \mid Q : T$.

(Estrut Res Res) Então $(\nu n\colon S)(\nu m\colon U)P \equiv (\nu m\colon U)(\nu n\colon S)P$, se $n \neq m$. É possível assumir por **(1)** que $\Gamma \vdash (\nu m\colon U)(\nu n\colon S)P : T$. Essa conclusão deve ter sido derivada duas vezes da regra de tipo (Proc Res) com as premissas $\Gamma, m : \textbf{\textit{Amb}}[Z], n : \textbf{\textit{Amb}}[V] \vdash P : T$, sendo que $S = \textbf{\textit{Amb}}[V]$ e $U = \textbf{\textit{Amb}}[Z]$. Aplicando o lema 5.3.2 conclui-se que $\Gamma, n : \textbf{\textit{Amb}}[V], m : \textbf{\textit{Amb}}[Z] \vdash P : T$. Aplicando (Proc Res) duas vezes nessa premissa conclui-se que $\Gamma \vdash (\nu n\colon S)(\nu m\colon U)P : T$.

(Estrut Res Par) Então $(\nu n\colon S)(P \mid Q) \equiv P \mid (\nu n\colon S)Q$ se $n \notin fn(P)$. É possível assumir por **(1)** que $\Gamma \vdash P \mid (\nu n\colon S)Q : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash P : T$ e **(b)** $\Gamma \vdash (\nu n\colon S)Q : T$ A premissa **(b)** deriva de (Proc Res) com **(c)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash Q : T$, sendo que $S = \textbf{\textit{Amb}}[V]$. Pelo lema 5.3.3 desde que $n \notin dom(\Gamma)$ conclui-se que **(d)** $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash P : T$. Aplicando (Proc Par) em **(d)** e **(c)** conclui-se $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash P \mid Q : T$. Aplicando (Proc Res) nessa conclusão $\Gamma \vdash (\nu n : \textbf{\textit{Amb}}[V])(P \mid Q) : T$.

(Estrut Res Amb) Então $(\nu n\colon S)m[P] \equiv m[(\nu n\colon S)P]$ se $n \neq m$. É possível assumir por **(2)** que $\Gamma \vdash m[(\nu n\colon S)P] : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Amb) com **(a)** $\Gamma \vdash m : \textbf{\textit{Amb}}[V]$ e **(b)** $\Gamma \vdash (\nu n\colon S) : V$, para algum V. A premissa

(b) deriva de (Proc Res) com **(c)** $\Gamma, n : \textbf{\textit{Amb}}[U] \vdash P : U$, sendo que $S = \textbf{\textit{Amb}}[U]$. Pelo lema 5.3.3 desde que $n \notin dom(\Gamma)$ conclui-se que **(d)** $\Gamma, n : \textbf{\textit{Amb}}[U] \vdash m : \textbf{\textit{Amb}}[V]$. Aplicando (Proc Amb) nas premissas **(c)** e **(d)** conclui-se $\Gamma, n : \textbf{\textit{Amb}}[U] \vdash m[P] : T$. Aplicando (Proc Res) nessa conclusão $\Gamma \vdash (\nu n : \textbf{\textit{Amb}}[U])m[P] : T$.

(Estrut Repl Par) Então $!P \equiv P \mid !P$. É possível assumir por **(2)** que $\Gamma \vdash P \mid !P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash P : T$ e $\Gamma \vdash !P : T$.

(Estrut Zero Par) Então $P \mid \mathbf{0} \equiv P$. É possível assumir por **(2)** que $\Gamma \vdash P : T$. Aplicando o lema 5.3.1 conclui-se $\Gamma \vdash \diamond$. Pela regra de tipo (Proc Zero) é possível concluir $\Gamma \vdash \mathbf{0} : T$. Aplicando (Proc Par) nessas premissas conclui-se $\Gamma \vdash P \mid \mathbf{0} : T$.

(Estrut Zero Repl) Então $!\mathbf{0} \equiv \mathbf{0}$. É possível assumir por **(2)** que $\Gamma \vdash \mathbf{0} : T$. Aplicando (Proc Repl) conclui-se $!\mathbf{0} : T$.

(Estrut Zero Ação) Então $\mathbf{0}.P \equiv P$. É possível assumir por **(1)** que $\Gamma \vdash P : T$. Aplicando o lema 5.3.1 conclui-se $\Gamma \vdash \diamond$. Pela regra de tipo (Proc Zero) é possível concluir $\Gamma \vdash \mathbf{0} : T$ Aplicando (Proc Ação) nas premissas conclui-se $\Gamma \vdash \mathbf{0}.P : T$, sendo que $T = \textbf{\textit{Cap}}[S]$ para algum S.

(Estrut Zero Res) Então $(\nu n\colon S)\mathbf{0} \equiv \mathbf{0}$. É possível assumir por **(2)** que $\mathbf{0} : T$. Pelo lema 5.3.3 é possível concluir que $\Gamma, n : \textbf{\textit{Amb}}[V] \vdash \mathbf{0} : T$, sendo que $S = \textbf{\textit{Amb}}[V]$. Aplicando (Proc Res) conlui-se $\Gamma \vdash (\nu n : \textbf{\textit{Amb}}[V])\mathbf{0} : T$.

(Estrut Ação) Então $\textbf{true}?M.P \equiv M.P$. É possível assumir por **(2)** que $\Gamma \vdash M.P : T$. Pelo lema 5.3.4 é possível concluir $\Gamma, \textbf{true} : Bool \vdash M.P : T$. Aplicando (Proc Contex) conclui-se $\Gamma \vdash \textbf{true}?M.P : T$, sendo que $T = \textbf{\textit{Cap}}[S]$ para algum S.

(Estrut Cont Ação), (Estrut Abs Proc) Simétrico ao caso **(1)**.

(Estrut Del) Então $del\ n.P \mid n[\mathbf{0}] \equiv P$. É possível assumir por **(2)** que $\Gamma \vdash P : T$. Aplicando o lema 5.3.3 duas vezes é possível concluir **(a)** $\Gamma, n : \textbf{\textit{Amb}}[S], \mathbf{0} : S \vdash P : T$. Aplicando o lema 5.3.5 é possível desmembrar as expressões **(b)** $\Gamma \vdash n : \textbf{\textit{Amb}}[S]$, **(c)**

$\Gamma \vdash \mathbf{0} : S$ e **(d)** $\Gamma \vdash P : T$. Aplicando (Proc Amb) nas premissas **(b)** e **(c)** conclui-se **(e)** $\Gamma \vdash n[\mathbf{0}] : V$. Aplicando (Proc Del) em **(e)** conclui-se **(f)** $\Gamma \vdash del\ n : T$, sendo que $T = \textbf{\textit{Cap}}[U]$. Aplicando (Proc Ação) em **(f)** e **(d)** conclui-se **(g)** $\Gamma \vdash del\ n.P : T$. Finalmente aplicando (Proc Par) em **(g)** e **(d)** conclui-se $del\ n.P \mid n[\mathbf{0}] : T$. □

5.3.8 Teorema (Subject Reduction)

Se $\Gamma \vdash P : T$ *e* $P \to Q$ *então* $\Gamma \vdash Q : T$

Prova. A prova é por indução na derivação de $P : T$. A cada passo da indução assumimos que a propriedade vale para todas as subderivações e procedemos pela análise do caso na regra final da derivação.

(Red Call Lc) Considere $P = x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid x\langle\tilde{z}\rangle$ e $Q = x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid P'\{\tilde{y} \leftarrow \tilde{z}\}$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : T$ e **(b)** $\Gamma \vdash x\langle\tilde{z}\rangle : T$, sendo que $T = \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(a)** deve ter sido derivada da regra de tipo (Proc Abs ε) com as premissas **(c)** $\Gamma, \tilde{y} : \tilde{W} \vdash x : \tilde{W}$ e **(d)** $\Gamma \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(b)** deve ter sido derivada da regra de tipo (Proc Cham ε) com as premissas **(e)** $\Gamma \vdash \tilde{z} : \tilde{W}$ e **(f)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}]$. O primeiro processo rodando em paralelo de Q é igual à **(a)**. Aplicando o lema 5.3.6 n vezes nas premissas **(c)**, **(d)** e **(e)** e a regra (Proc Par) conclui-se que $x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : T \mid P'\{\tilde{y} \leftarrow \tilde{z}\} : T$.

(Red Call S) Considere $P = n[Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] \mid m[n :: x\langle\tilde{z}\rangle \mid R]$ e $Q = n[Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] \mid m[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid R]$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash n[Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] : T$ e **(b)** $\Gamma \vdash m[n :: x\langle\tilde{z}\rangle \mid R] : T$, para algum T. A premissa **(a)** deve ter derivado da regra de tipo (Proc Amb) com **(c)** $\Gamma \vdash n : \textbf{\textit{Amb}}[S]$ e **(d)** $\Gamma \vdash Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : S$. A premissa **(d)** deriva de (Proc Par) com **(e)** $\Gamma \vdash Q' : S$ e **(f)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : S$. A premissa **(f)** deriva de (Proc Abs ε) com **(g)** $\Gamma, \tilde{y} : \tilde{W} \vdash x : \tilde{W}$ e **(h)** $\Gamma \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$, consequentemente S deve ser igual à $\textbf{\textit{Cap}}[\tilde{W}]$. Voltando ao segundo processo rodando em paralelo em P, a premissa **(b)** deve ter derivado da regra de tipo (Proc Amb) com **(i)** $\Gamma \vdash m : \textbf{\textit{Amb}}[U]$ e **(j)** $\Gamma \vdash n :: x\langle\tilde{z}\rangle \mid R : U$. A premissa **(j)** deriva de (Proc Par) com **(k)** $\Gamma \vdash n :: x\langle\tilde{z}\rangle : U$ e **(l)** $\Gamma \vdash R : U$. Completando as derivações de P, **(k)** deriva de (Proc Cham α) com **(m)**

$\Gamma \vdash \tilde{z} : V$, **(n)** $\Gamma \vdash x : \textbf{\textit{Abs}}[V]$ e **(o)** $\Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[V]]$, sendo que $(\alpha =::)$. Para a chamada do processo ser executado, V deve ser igual à $\tilde{W}$, e para as premissas **(c)** e **(o)** combinarem U deve ser igual a S. Todos os processos que compõem Q são iguais à P com exceção de **(k)**, que é onde ocorre a chamada da abstração de processo. Dessa forma, aplicando na ordem inversa as regras de tipos utilizadas para derivar P, mais o lema 5.3.6 n vezes nas premissas **(g)**, **(h)** e **(m)**, conclui-se que $n[Q \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] \mid m[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid R] : T$.

(Red Call U) Considere $P = n[Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid m[n \uparrow x\langle\tilde{z}\rangle \mid R]]$ e $Q = n[Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid m[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid R]]$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Amb) com as premissas **(a)** $\Gamma \vdash n : \textbf{\textit{Amb}}[T']$ e **(b)** $Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid m[n \uparrow x\langle\tilde{z}\rangle \mid R] : T'$, sendo que $T' = \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(b)** deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(c)** $\Gamma \vdash Q' : T'$ e **(d)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid m[n \uparrow x\langle\tilde{z}\rangle \mid R] : T'$. A premissa **(d)** novamente deve ter sido derivada de (Proc Par) gerando as nova premissas **(e)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : T'$ e **(f)** $\Gamma \vdash m[n \uparrow x\langle\tilde{z}\rangle \mid R] : T'$. A premissa **(e)** deriva de (Proc Abs ε) com **(g)** $\Gamma, \tilde{y} : \tilde{W} \vdash x : \tilde{W}$ e **(h)** $\Gamma \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(f)** deriva de (Proc Amb) com **(i)** $\Gamma \vdash m : \textbf{\textit{Amb}}[T'']$ e **(j)** $\Gamma \vdash n \uparrow x\langle\tilde{z}\rangle \mid R : T''$, sendo que $T'' = T'$. A premissa **(j)** deriva de (Proc Par) com **(k)** $\Gamma \vdash n \uparrow x\langle\tilde{z}\rangle : T''$ e **(l)** $\Gamma \vdash R : T''$. Completando as derivações de P, **(k)** deriva de (Proc Cham α) com as premissas **(m)** $\Gamma \vdash \tilde{z} : \tilde{W}$, **(n)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}]$ e **(o)** $\Gamma \vdash n : \textbf{\textit{Amb}}[T'']$, sendo que $(\alpha =\uparrow)$. Todos os processos que compõem Q são iguais à P com exceção de **(k)**, que é onde ocorre a chamada da abstração de processo. Dessa forma, aplicando na ordem inversa as regras de tipos utilizadas para derivar P, mais o lema 5.3.6 n vezes nas premissas **(g)**, **(h)** e **(m)**, conclui-se que $n[Q' \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' \mid m[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid R]] : T$.

(Red Call D) Considere $P = Q' \mid m \downarrow x\langle\tilde{z}\rangle \mid m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P']$ e $Q = Q' \mid P'\{\tilde{y} \leftarrow \tilde{z}\} \mid m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P']$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash Q' : T$ e **(b)** $\Gamma \vdash m \downarrow x\langle\tilde{z}\rangle \mid m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] : T$. A premissa **(b)** deriva novamente de (Proc Par) com **(c)** $\Gamma \vdash m \downarrow x\langle\tilde{z}\rangle : T$ e **(d)** $\Gamma \vdash m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] : T$. A premissa **(c)** deriva de (Proc Cham α) com **(e)** $\Gamma \vdash m : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[S]]$, **(f)** $\Gamma \vdash \tilde{z} : S$ e **(g)** $\Gamma \vdash x : \textbf{\textit{Abs}}[S]$, sendo que $\alpha =\downarrow$. Voltando ao

segundo processo rodando em paralelo em P, a premissa **(d)** deriva de (Proc Amb) com **(h)** $\Gamma \vdash m : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[S]]$ (m é o mesmo ambiente que já foi adicionado em Γ na premissa **(e)**, por isso deve ter o mesmo tipo), **(i)** $\Gamma \vdash R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : \textbf{\textit{Cap}}[S]$. Como ((**c**) e (**d**)) estão em paralelo, T deve ser igual a $\textbf{\textit{Cap}}[S]$. A premissa **(i)** deriva de (Proc Par) com **(j)** $\Gamma \vdash R : \textbf{\textit{Cap}}[S]$ e **(k)** $\Gamma \vdash x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P' : \textbf{\textit{Cap}}[S]$. Completando as derivações de P, **(k)** deriva de (Proc Abs ε) com **(l)** $\Gamma, \tilde{y} : \tilde{W} \vdash x : \tilde{W}$ e **(m)** $\Gamma \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$, consequentemente $S = \tilde{W}$. Todos os processos que compõem Q são iguais à P com exceção de **(c)**, que é onde ocorre a chamada da abstração de processo. Dessa forma, aplicando na ordem inversa as regras de tipos utilizadas para derivar P, mais o lema 5.3.6 n vezes nas premissas **(l)**, **(m)** e **(f)**, conclui-se que $Q' \mid P'\{\tilde{y} \leftarrow \tilde{z}\} \mid m[R \mid x : \textbf{\textit{Abs}}[\tilde{W}] \triangleright (\tilde{y} : \tilde{W}).P'] : T$.

(Red Com Lc) Considere $P = (\tilde{y} : \tilde{W}).P' \mid \langle \tilde{z} \rangle.Q'$ e $Q = P'\{\tilde{y} \leftarrow \tilde{z}\} \mid Q'$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash (\tilde{y} : \tilde{W}).P' : T$ e **(b)** $\Gamma \vdash \langle \tilde{z} \rangle.Q' : T$. A premissa **(a)** deriva de (Proc Entrada ε) com **(c)** $\Gamma, \tilde{y} : \tilde{W} \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$, consequentemente $T = \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(b)** deriva de (Proc Ação) com **(e)** $\Gamma \vdash \langle \tilde{z} \rangle : \textbf{\textit{Cap}}[\tilde{W}]$ e **(f)** $Q' : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(e)** deriva de (Proc Saída ε) com **(g)** $\Gamma \vdash \tilde{z} : \tilde{W}$. Aplicando o lema 5.3.6 n vezes nas premissas **(c)** e **(g)**, mais a regra (Proc Par) conclui-se que $P'\{\tilde{y} \leftarrow \tilde{z}\} \mid Q' : T$.

(Red Com S) Considere $P = n[m :: (\tilde{y} : \tilde{W}).P'.\mathbf{0} \mid Q'] \mid m[n :: \langle \tilde{z} \rangle.R \mid S]$ e $Q = n[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid Q'] \mid m[R \mid S]$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash n[m :: (\tilde{y} : \tilde{W}).P'.\mathbf{0} \mid Q'] : T$ e **(b)** $\Gamma \vdash m[n :: \langle \tilde{z} \rangle.R \mid S] : T$, para algum T. A premissa **(a)** deriva de (Proc Amb) com **(c)** $\Gamma \vdash n : \textbf{\textit{Amb}}[S]$ e **(d)** $\Gamma \vdash m :: (\tilde{y} : \tilde{W}).P'.\mathbf{0} \mid Q' : S$. A premissa **(d)** deriva de (Proc Par) com **(e)** $\Gamma \vdash m :: (\tilde{y} : \tilde{W}).P'.\mathbf{0} : S$ e **(f)** $\Gamma \vdash Q' : S$. A premissa **(e)** deriva de (Proc Ação) com **(h)** $\Gamma \vdash m :: (\tilde{y} : \tilde{W}).P' : \textbf{\textit{Cap}}[V]$ e **(i)** $\Gamma \vdash \mathbf{0} : \textbf{\textit{Cap}}[V]$. A premissa **(h)** deriva de (Proc Entrada α) com **(j)** $\Gamma, \tilde{y} : \tilde{W} \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$ e **(k)** $\Gamma \vdash m : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}]]$, sendo que $(\alpha =::)$. Consequentemente $S = \textbf{\textit{Cap}}[\tilde{W}]$ e $V = \tilde{W}$. Voltando ao segundo processo rodando em paralelo em P, a premissa **(b)** deve ter derivado da regra de tipo (Proc Amb) com **(l)** $\Gamma \vdash m : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}]]$ (conforme premissa **(k)**) e **(m)** $\Gamma \vdash n :: \langle \tilde{z} \rangle.R \mid S : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(m)** deriva de (Proc Par) com $\Gamma \vdash n :: \langle \tilde{z} \rangle.R : \textbf{\textit{Cap}}[\tilde{W}]$ e **(n)** $\Gamma \vdash S : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(n)** deriva de (Proc Ação)

com **(p)** $\Gamma \vdash n :: \langle \tilde{z} \rangle : \textbf{\textit{Cap}}[\tilde{W}]$ e **(q)** $\Gamma \vdash R : \textbf{\textit{Cap}}[\tilde{W}]$. Completando as derivações de P, **(p)** deriva de (Proc Saída α) com **(r)** $\Gamma \vdash \tilde{z} : \tilde{W}$ e **(s)** $\Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}]]$ (conforme premissa **(c)**). Para concluir $Q : T$, deve-se utilizar todas as premissas derivadas e montar a expressão. Aplicando o lema 5.3.6 n vezes nas premissas **(j)** e **(r)**, mais a regra (Proc Par) em **(f)** gera **(t)** $\Gamma \vdash P'\{\tilde{y} \leftarrow \tilde{z}\} \mid Q' : S$. Aplicando (Proc Amb) em **(t)** e **(c)** gera **(u)** $\Gamma \vdash n[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid Q'] : T$. Aplicando (Proc Par) em **(q)** e **(n)** gera **(v)** $\Gamma \vdash R \mid S : \textbf{\textit{Cap}}[\tilde{W}]$. Aplicando (Proc Amb) em **(v)** gera **(x)** $\Gamma \vdash m[R \mid S] : T$ e (Proc Par) em **(u)** e **(x)** conclui-se $\Gamma \vdash n[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid Q'] \mid m[R \mid S] : T$.

(Red Com R1) Considere $P = u[n \downarrow (\tilde{y} : \tilde{W}).P'.\mathbf{0} \mid n[u \uparrow \langle \tilde{z} \rangle.Q' \mid R] \mid S]$ e $Q = u[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid n[Q' \mid R] \mid S]$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Amb) com as premissas **(a)** $\Gamma \vdash u : \textbf{\textit{Amb}}[U]$ e **(b)** $\Gamma \vdash n \downarrow (\tilde{y} : \tilde{W}).P'.\mathbf{0} \mid n[u \uparrow \langle \tilde{z} \rangle.Q' \mid R] \mid S : U$. A premissa **(b)** deriva (Proc Par) com **(c)** $\Gamma \vdash n \downarrow (\tilde{y} : \tilde{W}).P'.\mathbf{0} : U$ e **(d)** $\Gamma \vdash n[u \uparrow \langle \tilde{z} \rangle.Q' \mid R] \mid S : U$. A premissa **(c)** deriva de (Proc Ação) com **(f)** $\Gamma \vdash n \downarrow (\tilde{y} : \tilde{W}).P' : \textbf{\textit{Cap}}[S]$ e **(g)** $\Gamma \vdash \mathbf{0} : \textbf{\textit{Cap}}[S]$. A premissa **(f)** deriva de (Proc Entrada α) com **(h)** $\Gamma, \tilde{y} : \tilde{W} \vdash P' : \textbf{\textit{Cap}}[\tilde{W}]$ e **(i)** $\Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}]]$, sendo que $(\alpha = \downarrow)$, consequentemente S é igual à $\tilde{W}$ (para preencher o tipo correto da premissa **(f)**). Voltando ao segundo processo rodando em paralelo em P, **(d)** deriva de (Proc Par) com **(j)** $\Gamma \vdash n[u \uparrow \langle \tilde{z} \rangle.Q' \mid R] : U$ e **(k)** $\Gamma \vdash S : U$. A premissa **(j)** deriva de (Proc Amb) com **(l)** $\Gamma \vdash n : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}]]$ (conforme premissa **(i)**) e **(m)** $\Gamma \vdash u \uparrow \langle \tilde{z} \rangle.Q' \mid R : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(m)** deriva de (Proc Par) com **(n)** $\Gamma \vdash u \uparrow \langle \tilde{z} \rangle.Q' : \textbf{\textit{Cap}}[\tilde{W}]$ e **(o)** $\Gamma \vdash R : \textbf{\textit{Cap}}[\tilde{W}]$. A premissa **(n)** deriva de (Proc Ação) com **(q)** $\Gamma \vdash u \uparrow \langle \tilde{z} \rangle : \textbf{\textit{Cap}}[\tilde{W}]$ e **(r)** $\Gamma \vdash Q' : \textbf{\textit{Cap}}[\tilde{W}]$. Finalmente, a premissa **(q)** deriva de (Proc Saída α) com as premissas **(s)** $\Gamma \vdash \tilde{z} : \tilde{W}$ e **(t)** $\Gamma \vdash u : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[\tilde{W}]]$, consequentemente $U = \textbf{\textit{Cap}}[\tilde{W}]$ (para preencher o tipo correto da premissa **(a)**). Para concluir $Q : T$, deve-se utilizar todas as premissas derivadas e montar a expressão. Aplicando (Proc Par) em **(r)** e **(o)** gera **(u)** $\Gamma \vdash Q' \mid R : \textbf{\textit{Cap}}[\tilde{W}]$. Aplicando (Proc Amb) em **(i)** e **(u)** mais (Proc Par) em **(t)** gera **(v)** $\Gamma \vdash n[Q' \mid R] \mid S : U$. Aplicando o lema 5.3.6 n vezes nas premissas **(h)** e **(s)** mais (Proc Par) em **(v)** mais (Proc Amb) no resultado com a premissa **(a)** conclui-se que $u[P'\{\tilde{y} \leftarrow \tilde{z}\} \mid n[Q' \mid R] \mid S] : T$.

(Red Com R2) Semelhante a prova **(Red Com R1)**. Somente está invertido o ambiente onde está o processo que recebe um valor e o ambiente que envia a mensagem, portanto

as derivações e a prova são semelhantes.

(Red In) Considere $P = n[in\ m.P' \mid Q'] \mid m[R]$ e $Q = m[n[P' \mid Q'] \mid R]$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Par) com as premissas **(a)** $\Gamma \vdash n[in\ m.P' \mid Q'] : T$ e **(b)** $\Gamma \vdash m[R] : T$. A premissa **(a)** deriva de (Proc Amb) com **(c)** $\Gamma \vdash n : \textbf{\textit{Amb}}[S]$ e **(d)** $\Gamma \vdash in\ m.P' \mid Q' : S$. A premissa **(d)** deriva de (Proc Par) com **(e)** $\Gamma \vdash in\ m.P' : S$ e **(f)** $\Gamma \vdash Q' : S$. A premissa **(e)** de (Proc Ação) com **(h)** $\Gamma \vdash in\ m : S$ e **(i)** $\Gamma \vdash P' : S$, sendo que $S = \textbf{\textit{Cap}}[V]$, para algum V. A premissa **(h)** deriva de (Exp *in*) com **(j)** $\Gamma \vdash m : \textbf{\textit{Amb}}[U]$. Voltando a segunda expressão rodando em paralelo em Q, a premissa **(b)** deriva de (Proc Amb) com **(k)** $\Gamma \vdash m : \textbf{\textit{Amb}}[U]$ (conforme premissa **(j)**) e **(l)** $\Gamma \vdash R : U$. Para concluir $Q : T$, deve-se utilizar as premissas derivadas e montar a expressão. Aplicando (Proc Par) em **(i)** e **(f)** gera **(m)** $P' \mid Q' : S$. Aplicando (Proc Amb) em **(c)** e **(m)** gera **(n)** $\Gamma \vdash n[P' \mid Q'] : U$. Aplicando (Proc Par) em **(n)** e **(l)** gera **(o)** $\Gamma \vdash n[P' \mid Q'] \mid R : U$. Finalmente aplicando (Proc Amb) em **(j)** e **(o)** conclui-se $\Gamma \vdash m[n[P' \mid Q'] \mid R] : T$.

(Red Out) Considere $P = m[n[out.P' \mid Q'] \mid R]$ e $Q = n[P' \mid Q'] \mid m[R]$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter sido derivada da regra de tipo (Proc Amb) com as premissas **(a)** $\Gamma \vdash m : \textbf{\textit{Amb}}[S]$ e **(b)** $\Gamma \vdash n[out.P' \mid Q'] \mid R : S$. A premissa **(b)** deriva de (Proc Par) com **(c)** $\Gamma \vdash n[out.P' \mid Q'] : S$ e **(d)** $\Gamma \vdash R : S$. A premissa **(c)** deriva de (Proc Amb) com **(e)** $\Gamma \vdash n : \textbf{\textit{Amb}}[U]$ e **(f)** $\Gamma \vdash out.P' \mid Q' : U$. A premissa **(f)** deriva de (Proc Par) com **(g)** $\Gamma \vdash out.P' : U$ e **(h)** $\Gamma \vdash Q' : U$. A premissa **(g)** deriva de (Proc Ação) com **(j)** $\Gamma \vdash out : U$ e **(k)** $\Gamma \vdash P : U$, sendo que $U = \textbf{\textit{Cap}}[V]$ para algum V. Para concluir $Q : T$, deve-se utilizar as premissas derivadas e montar a expressão. Aplicando (Proc Par) em **(k)** e **(h)** gera **(l)** $\Gamma \vdash P' \mid Q' : U$. Aplicando (Proc Amb) em **(e)** e **(l)** gera **(m)** $\Gamma \vdash n[P' \mid Q'] : T$. Aplicando (Proc Amb) em **(a)** e **(d)** gera **(n)** $\Gamma \vdash m[R] : T$. Finalmente, aplicando (Proc Par) em **(m)** e **(n)** conclui-se que $\Gamma \vdash n[P' \mid Q'] \mid m[R] : T$.

(Red Res) Considere $P = (\nu n : S)P'$ e $Q = (\nu n : S)Q'$, sendo que $P' \rightarrow Q'$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter derivada da regra de tipo (Proc Res) com a premissa **(a)** $\Gamma, n : \textbf{\textit{Amb}}[U] \vdash P' : T$, onde $S = \textbf{\textit{Amb}}[U]$. Pelo teorema 5.3.8 é possível concluir que $Q' : T$, se $P' \rightarrow Q'$. Portanto $\Gamma, n : \textbf{\textit{Amb}}[U] \vdash Q' : T$. Aplicando

(Proc Res) nessa premissa conclui-se que $(\nu n : S)Q^{'} : T$.

(Red Amb) Considere $P = n[P^{'}]$ e $Q = n[Q^{'}]$, sendo que $P^{'} \rightarrow Q^{'}$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter derivada da regra de tipo (Proc Amb) com as premissas **(a)** $\Gamma \vdash n :$ ***Amb***$[S]$ e **(b)** $\Gamma \vdash P^{'} : S$ para algum S. Pelo teorema 5.3.8 é possível concluir que $Q^{'} : S$, se $P^{'} \rightarrow Q^{'}$. Portanto $\Gamma \vdash Q^{'} : S$. Aplicando (Proc Amb) nas premissas conclui-se que $n[Q^{'}] : T$.

(Red Par) Considere $P = P^{'} \mid R$ e $Q = Q^{'} \mid R$, sendo que $P^{'} \rightarrow Q^{'}$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter derivada da regra de tipo (Proc Par) com as premissas $\Gamma \vdash P^{'} : T$ e $\Gamma \vdash R : T$. Pelo teorema 5.3.8 é possível concluir que $Q^{'} : T$, se $P^{'} \rightarrow Q^{'}$. Portanto $\Gamma \vdash Q^{'} : T$. Aplicando (Proc Par) nas premissa conclui-se que $Q^{'} \mid R : T$.

(Red $\equiv$) Então $P \equiv P^{'}$ e $Q \equiv Q^{'}$, sendo que $P^{'} \rightarrow Q^{'}$. Assume-se pelo teorema 5.3.8 que $P : T$. Pelo teorema 5.3.7 é possível concluir que $P^{'} : T$. Pelo teorema 5.3.8 é possível concluir que $Q^{'} : T$, se $P^{'} \rightarrow Q^{'}$. Portanto $\Gamma \vdash Q^{'} : T$. Novamente pelo teorema 5.3.7 é possível concluir que $Q : T$.

(Red Cont) Considere $P = C(P^{'})$ e $Q = C(Q^{'})$, sendo que $P^{'} \rightarrow Q^{'}$. Assume-se pelo teorema 5.3.8 que $P : T$. Essa conclusão deve ter derivada da regra de tipo (Exp Cont) com a premissa $\Gamma \vdash P : S$ para algum S. Pelo teorema 5.3.8 é possível concluir que $Q^{'} : S$, se $P^{'} \rightarrow Q^{'}$. Portanto $\Gamma \vdash Q^{'} : S$. Aplicando (Exp Cont) nessa premissa conclui-se que $C(Q^{'}) : T$.

(Red Guard) Então $C(k?M.P) \rightarrow C^{'}(P)$ se $C \models k$ e se $C(M.P) \rightarrow C^{'}(P)$. NEssa redução a avaliação de C para $C^{'}$ leva em consideração que a capacidade M pode ser a capacidade in ou out. Nesse caso o ambiente que exerceu a capacidade mudará de localização na hierarquia, fazendo com o espaço reservado $\odot$ também mude de localização. Essa redução também leva em consideração que $k?M.P$ é equivalente a $M.P$ se a expressão de contexto k satisfaz o contexto C, ou seja, **true**$?M.P$, e que ambas as expressões avaliam para P. Nesse caso, essa regra equivale a congruência **(Estrut Ação)** já provada. Portanto a prova dessa redução segue a mesma lógica de **(Red Cont)**. □

5.4 Resumo do capítulo

Este capítulo propôs um sistema de tipos para o CASC, descrito na Seção 4.4, com foco no controle de comunicação entre processos. Algumas alterações na sintaxe original do CASC foram necessárias para o desenvolvimento do sistema de tipos. A principal alteração foi a exclusão das primitivas que permitiam enviar mensagens a qualquer ambiente, independente do nome. Consequentemente, foram alteradas a congruência estrutural e algumas regras de reduções de processos. Como estudo de caso, foram modelados dois cenários demonstrando o uso do CASC e do sistema de tipos. Ao final, a propriedade *preservation* (ou *subject reduction*) do sistema de tipos foi provada formalmente, demostrando que o sistema de tipos está correto.

6 CONCLUSÃO

A computação pervasiva trouxe uma série de novos desafios para o desenvolvimento de aplicações. Um dos principais desafios é a programação levando em consideração a sensibilidade ao contexto e a mobilidade de código entre dispositivos. Existem muitas propostas na literatura para facilitar o desenvolvimento de aplicações. A maioria das propostas consiste em adicionar funcionalidades em linguagens existentes ou criar novos paradigmas para permitir desenvolver com maior facilidade aplicações pervasivas. A maioria das propostas pesquisadas atinge o objetivo final, porém são soluções baseadas em linguagens que foram desenvolvidas para programação de ambientes estáticos. Dessa forma, a programação nessas linguagens não é feita de uma forma natural.

Alguns autores argumentam que, com o novo paradigma pervasivo, são necessários novos formalismos que permitam descrever de uma forma simples e natural cenários pervasivos. Modelos formais podem ser utilizados para especificação a de linguagens de programação. Além disso, podem ser utilizados para descrever em qualquer nível de detalhe, um sistema que será desenvolvido. Um modelo formal amplamente utilizado para especificação de linguagens de programação é o Cálculo-λ, que pode ser visto como uma simples linguagem de programação, na qual qualquer computação é um objeto matemático, no qual várias demonstrações podem ser provadas. Nesse sentido esta dissertação estudou o uso do Cálculo de Ambientes Sensível ao Contexto (CASC), uma das muitas variantes do Cálculo de Ambientes (um modelo formal para especificação de sistemas móveis), como base para descrever ambientes pervasivos.

A principal contribuição deste trabalho foi a definição de um sistema de tipos para o CASC, visando o controle de comunicação. Com esse sistema de tipos, é possível restringir a comunicação dentro dos ambientes, ou seja, é possível conhecer previamente o tipo de troca que será permitido aos processos rodando dentro do ambiente. A vantagem é a segurança para o programador, pois o *typechecker* poderá garantir que a comunicação será feita de forma correta entre mensageiro e receptor.

O sistema de tipos desenvolvido força que o tipo do ambiente seja definido estaticamente, não sendo mais possível alterá-lo dinamicamente. Conforme as regras de redução apresentadas na Figura 5.6, esse comportamento está correto, pois o tipo não precisa ser alterado. Porém, em aplicações pervasivas, que são altamente dinâmicas, poderá ser necessário permitir que o ambiente altere o seu tipo dinamicamente. Essa questão é abordada na parte de trabalhos

futuros.

Por fim, foram modelados dois cenários utilizando o CASC, demonstrando o uso do cálculo e do sistema de tipos. A propriedade *preservation* do sistema de tipos foi provada formalmente, demostrando que o sistema de tipos está correto, ou seja, atingindo o objetivo principal do trabalho. Como trabalhos futuros, é possível sugerir:

- Adicionar o controle de mobilidade no sistema de tipos, conforme proposto no CA original (Seção 4.2.2).

- Estudar a viabilidade de se utilizar tipos dependentes (Seção 3.6) ou TRD (Seção 3.6.1) no CASC. Tipos dependentes permitem um controle mais dinâmico de tipos, porém são mais complexos de se trabalhar. Existe uma proposta na literatura de usar tipos dependentes no CA (LHOUSSAINE; SASSONE, 2004).

- Desenvolver uma forma mais simples de enviar nomes de ambientes como mensagem. Este foi um problema apresentado durante a modelagem do cenário 2 do estudo de caso.

- Adicionar inferência de tipos (FUH; MISHRA, 1990) no sistema onde for possível. Ou seja, tornar implícito o tipo do termo e o sistema de tipos descobrir o tipo correto do termo através da análise da expressão.

- Permitir que um ambiente possa alterar o seu tipo dinamicamente, ou permitir que um mesmo ambiente possa trocar mensagens com tipos diferentes, por exemplo, um ambiente com dois processos rodando em paralelo. Um processo possui tipo de troca T e outro S. Permitir que esses dois processos consigam rodar em paralelo. Uma solução seria utilizar tipos variáveis (Seção 3.4.4), que é uma estrutura que permite representar um conjunto de variáveis com tipos diferentes.

- Refinar os tipos propostos adicionando os valores (possíveis resultados finais de avaliações dos termos). Após a definição, provar a propriedade *progress* do sistema de tipos.

REFERÊNCIAS

ABOWD, G. D. et al. Towards a Better Understanding of Context and Context-Awareness. In: HUC '99: PROCEEDINGS OF THE 1ST INTERNATIONAL SYMPOSIUM ON HANDHELD AND UBIQUITOUS COMPUTING, London, UK. **Anais...** Springer Berlin / Heidelberg, 1999. p.304–307. (Lecture Notes in Computer Science, v.1707).

ALTENKIRCH, T.; MCBRIDE, C.; MCKINNA, J. **Why Dependent Types Matter**. [S.l.: s.n.], 2005. Manuscrito, Disponível em <http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf>. Acesso em: maio 2011.

ALY, S. G.; NADI, S.; HAMDAN, K. A Java-Based Programming Language Support of Location Management in Pervasive Systems. **International Journal of Computer Science and Network Security**, [S.l.], v.8, n.6, p.329–336, 2008.

ARAUJO, R. B. de. Computação Ubíqua: princípios, tecnologias, serviços e desafios. In: XXI SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES (SBRC), Natal - RN. **Anais...** SBC, 2003. p.45–115.

AUGUSTIN, I. **Abstrações para uma linguagem de programação visando aplicações móveis em um ambiente de *Pervasive Computing***. 2004. Tese (Doutorado) — Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil.

AUGUSTO, J. C. Ambient Intelligence : the confluence of ubiquitous / pervasive computing and artificial intelligence. In: SCHUSTER, A. J. (Ed.). **Intelligent Computing Everywhere**. London - UK: Springer, 2007. p.213–234.

BUGLIESI, M.; CASTAGNA, G.; CRAFA, S. Access control for mobile agents: the calculus of boxed ambients. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, New York, NY, USA, v.26, p.57–124, January 2004.

CARDELLI, L. **Phase Distinctions in Type Theory**. Palo Alto, CA, USA: Digital Equipment Corporation, 1988. Não publicado, Disponível em <http://lucacardelli.name/Papers/PhaseDistinctions.A4.pdf>. Acesso em: maio 2011.

CARDELLI, L. Type Systems. **ACM Computing Surveys (CSUR)**, New York, NY, USA, v.28, p.263–264, 1996.

CARDELLI, L. Type Systems. In: TUCKER, A. B. (Ed.). **The Computer Science and Engineering Handbook**. 2nd.ed. [S.l.]: CRC Press, 2004.

CARDELLI, L.; GORDON, A. D. Mobile Ambients. In: FIRST INTERNATIONAL CONFERENCE ON FOUNDATIONS OF SOFTWARE SCIENCE AND COMPUTATION STRUCTURE, London, UK. **Proceedings...** Springer-Verlag, 1998. v.1378, p.140–155.

CARDELLI, L.; GORDON, A. D. Types for mobile ambients. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 26., New York, NY, USA. **Proceedings...** ACM, 1999. p.79–92. (POPL '99).

CARDELLI, L.; GORDON, A. D. Mobile ambients. **Theoretical Computer Science**, Essex, UK, v.240, p.177–213, June 2000.

CARDELLI, L.; GORDON, A. D.; GHELLI, G. Mobility Types for Mobile Ambients. In: INTERNATIONAL COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, 26., London, UK. **Proceedings...** Springer-Verlag, 1999. p.230–239. (ICAL '99).

CARDELLI, L.; WEGNER, P. On understanding types, data abstraction, and polymorphism. **ACM Computing Surveys (CSUR)**, New York, NY, USA, v.17, p.471–523, 1985.

CHARATONIK, W.; GORDON, A.; TALBOT, J.-M. Finite-Control Mobile Ambients. In: LE MÉTAYER, D. (Ed.). **Programming Languages and Systems**. [S.l.]: Springer Berlin / Heidelberg, 2002. p.295–313. (Lecture Notes in Computer Science, v.2305).

CHURCH, A. A Formulation of the Simple Theory of Types. **Journal of Symbolic Logic**, [S.l.], v.5, n.2, p.56–68, 1940.

COPPO, M. et al. M^3: mobility types for mobile processes in mobile ambients. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.78, n.0, p.144 – 177, 2003. CATS'03, Computing: the Australasian Theory Symposium.

DAPOIGNY, R.; BARLATIER, P. Towards a Context Theory for Context-aware systems. In: PROCEEDING OF THE 2007 CONFERENCE ON ADVANCES IN AMBIENT INTELLIGENCE, Amsterdam, The Netherlands, The Netherlands. **Anais...** IOS Press, 2007. p.36–55.

DAPOIGNY, R.; BARLATIER, P. **Towards a Conceptual Structure Based on Type theory**. 2009. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.8135>. Acesso em: maio 2011.

DERANSART, P.; SMAUS, J.-G. Subject Reduction of Logic Programs as Proof-Theoretic Property. **Journal of Functional and Logic Programming**, [S.l.], v.2002, 2002.

FARMER, W. M. The seven virtues of simple type theory. **Journal of Applied Logic**, [S.l.], v.6, n.3, p.267–286, 2008.

FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding Code Mobility. **IEEE Transactions on Software Engineering**, Los Alamitos, CA, USA, v.24, n.5, p.342–361, 1998.

FUH, Y.-C.; MISHRA, P. Type inference with subtypes. **Theoretical Computer Science**, [S.l.], v.73, n.2, p.155 – 175, 1990.

GORDON, A. **Types for Mobile Ambients**. Slides presentation on APPSEM'98., 1998. Disponível em: <http://www.disi.unige.it/appsem98/proceedings/gordon_slides.pdf.gz>. Acesso em: agosto 2011.

GORDON, A. D. **Functional programming and input/output**. New York, NY, USA: Cambridge University Press, 1994. 171p.

GRIMM, R. et al. Systems directions for pervasive computing. In: EIGHTH WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2001. p.147–151.

HENNESSY, M.; INGÓLFSDÓTTIR, A. A theory of communicating processes with value passing. **Information and Computation**, Duluth, MN, USA, v.107, p.202–236, December 1993.

HENRICKSEN, K.; INDULSKA, J. A Software Engineering Framework for Context-Aware Pervasive Computing. In: IEEE CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATIONS (PERCOM), 2., Orlando, Florida, USA. **Proceedings...** [S.l.: s.n.], 2004. p.77–86.

HOFER, T. et al. Context-Awareness on Mobile Devices - the Hydrogen Approach. In: HICSS '03. PROCEEDINGS OF THE 36TH ANNUAL HAWAII INTERNATIONAL CONFERENCE

ON SYSTEM SCIENCES, Washington, DC, USA. **Anais...** IEEE Computer Society, 2003. p.292–302.

HUANG, S.; MANGS, J. Pervasive Computing: migrating to mobile devices: a case study. In: ANNUAL IEEE SYSTEMS CONFERENCE, 2., Montreal, Quebec, Canada. **Anais...** IEEE Computer Society, 2008. p.1–8.

HUTH, M.; RYAN, M. **Lógica em Ciência da Computação**: modelagem e argumentação sobre sistemas. 2.ed. Rio de Janeiro: LTC, 2008. 326p.

JUNBIN, Z. et al. A Table-Driven Programming Paradigm for Context-Aware Application Development. **IEEE/IPSJ International Symposium on Applications and the Internet**, Los Alamitos, CA, USA, p.121–124, 2009.

KJÆRGAARD, M. B.; BUNDE-PEDERSEN, J. Towards a Formal Model of Context Awareness. In: FIRST INTERNATIONAL WORKSHOP ON COMBINING THEORY AND SYSTEMS BUILDING IN PERVASIVE COMPUTING (PERVASIVE 2006), Dublin, Ireland. **Proceedings...** [S.l.: s.n.], 2006. p.667–674.

LEE, S.; PARK, S.; LEE, S.-g. A Study on Issues in Context-Aware Systems Based on a Survey and Service Scenarios. In: ACIS INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ARTIFICIAL INTELLIGENCES, NETWORKING AND PARALLEL/DISTRIBUTED COMPUTING, 2009., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.8–13. (SNPD '09).

LEVI, F.; SANGIORGI, D. Mobile safe ambients. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, New York, NY, USA, v.25, p.1–69, January 2003.

LHOUSSAINE, C.; SASSONE, V. A Dependently Typed Ambient Calculus. In: SCHMIDT, D. (Ed.). **Programming Languages and Systems**. [S.l.]: Springer Berlin / Heidelberg, 2004. p.171–187. (Lecture Notes in Computer Science, v.2986).

LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, New York, NY, USA, v.16, p.1811–1841, November 1994.

LUO, Z. Dependent record types revisited. In: WORKSHOP ON MODULES AND LIBRARIES FOR PROOF ASSISTANTS, 1., New York, NY, USA. **Proceedings...** ACM, 2009. p.30–37. (MLPA '09).

MARGARIA, I.; ZACCHI, M. Access control in mobile ambient calculi: a comparative view. **Theoretical Computer Science**, Essex, UK, v.398, p.183–202, May 2008.

MARTIN-LÖF, P. **An Intuitionistic Theory of Types**. [S.l.]: University of Stockholm, 1972.

MILNER, R. **Communicating and mobile systems**: the π-calculus. New York, NY, USA: Cambridge University Press, 1999. 174p.

MILNER, R.; PARROW, J.; WALKER, D. A calculus of mobile processes, I and II. **Information and Computation**, [S.l.], v.100, n.1, p.1 – 77, 1992.

MIN, X. et al. Isotope Programming Model: a kind of program model for context-aware application. In: MUE '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON MULTIMEDIA AND UBIQUITOUS ENGINEERING, Washington, DC, USA. **Anais...** IEEE Computer Society, 2007. p.597–602.

MOSTÉFAOUI, G. K.; PASQUIER-ROCHA, J.; BRÉZILLON, P. Context-Aware Computing: a guide for the pervasive computing community. **ICPS '04: IEEE/ACS International Conference on Pervasive Services**, Los Alamitos, CA, USA, p.39–48, 2004.

NIEMELA, E.; LATVAKOSKI, J. Survey of requirements and solutions for ubiquitous software. In: MUM '04: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON MOBILE AND UBIQUITOUS MULTIMEDIA, New York, NY, USA. **Anais...** ACM, 2004. p.71–78.

NORELL, U. **Towards a practical programming language based on dependent type theory**. 2007. Tese (Doutorado) — Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

PIERCE, B. C. Foundational Calculi for Programming Languages. In: TUCKER, A. B. (Ed.). **The Computer Science and Engineering Handbook**. [S.l.]: CRC Press, 1997. p.2190–2207.

PIERCE, B. C. **Types and Programming Languages**. Cambridge, Massachusetts: MIT Press, 2002. 645p.

POKKUNURI, B. P. Object Oriented Programming. **ACM SIGPLAN Notices**, New York, NY, USA, v.24, n.11, p.96–101, Nov. 1989.

QIN, Z.; ZHENG, X.; XING, J. **Software Architecture**. [S.l.]: Springer eBooks, 2008. 400p. (Advanced Topics in Science and Technology in China).

RARAU, A.; BENTA, K. I.; CREMENE, M. Multifaceted based language for pervasive services with deterministic and fully defined behavior. **International Conference on Pervasive Services**, Los Alamitos, CA, USA, p.76–79, 2007.

RUAS, C. Internet sem fio para passageiros. **Correio do Povo**, [S.l.], n.183, Ano 115, quinta-feira, 1º Abril 2010.

SAHA, D.; MUKHERJEE, A. Pervasive Computing: a paradigm for the 21st century. **Computer**, Los Alamitos, CA, USA, v.36, p.25–31, 2003.

SANGIORGI, D. Extensionality and intensionality of the ambient logics. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 28., New York, NY, USA. **Proceedings...** ACM, 2001. p.4–13. (POPL '01).

SANGIORGI, D.; WALKER, D. π**-Calculus**: a theory of mobile processes. New York, NY, USA: Cambridge University Press, 2001. 596p.

SATYANARAYANAN, M. Pervasive Computing: vision and challenges. **IEEE Personal Communications**, [S.l.], v.8, n.4, p.10–17, Aug. 2001.

SEBESTA, R. W. **Concepts of Programming Languages**. 9th.ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2009. 696p.

SETZER, A. Object-Oriented Programming in Dependent Type Theory. In: NILSSON, H. (Ed.). **Trends in Functional Programming**. The University of Chicago Press, IL 60637, USA: Intellect Books, 2007. v.7, p.91–108.

SIEWE, F.; CAU, A.; ZEDAN, H. CCA: a calculus of context-aware ambients. In: IEEE 23RD INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS WORKSHOPS, 2009. WAINA '09., Bradford, United Kingdom. **Proceedings...** IEEE Computer Society, 2009. p.972 –977.

SIEWE, F.; ZEDAN, H.; CAU, A. The Calculus of Context-aware Ambients. **Journal of Computer and System Sciences**, Orlando, FL, USA, v.77, p.597–620, July 2011.

THOMPSON, S. **Type Theory and Functional Programming**. [S.l.]: Addison-Wesley, 1991. 388p.

WAHLSTEDT, D. **Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion**. 2007. Tese (Doutorado) — Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

WANT, R.; PERING, T. System challenges for ubiquitous & pervasive computing. In: ICSE '05: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York, NY, USA. **Anais...** ACM, 2005. p.9–14.

WATT, D. A. **Programming Language Design Concepts**. West Sussex, England: John Wiley & Sons, 2004. 492p. With contributions by William Findlay.

WEIS, T.; BECKER, C.; BRANDLE, A. Towards a programming paradigm for pervasive applications based on the ambient calculus. In: INTERNATIONAL WORKSHOP ON COMBINING THEORY AND SYSTEMS BUILDING IN PERVASIVE COMPUTING, Dublin, Ireland. **Anais...** [S.l.: s.n.], 2006. co-located with Pervasive 2006.

WEISER, M. The Computer for the 21st Century. **Scientific American**, [S.l.], v.265, n.3, p.94–104, September 1991.

WING, J. M. **FAQ on π-Calculus**. Microsoft Internal Memo. Acesso em: abril 2012.

WRIGHT, A. Type theory comes of age. **Communications of the ACM**, New York, NY, USA, v.53, n.2, p.16–17, 2010.

WRIGHT, A. K.; FELLEISEN, M. A syntactic approach to type soundness. **Information and Computation**, Duluth, MN, USA, v.115, p.38–94, November 1994.

YAO, Q. et al. New Programming Model for Pervasive Computing. In: ICEBE '08. IEEE INTERNATIONAL CONFERENCE ON E-BUSINESS ENGINEERING, Xi'an, China. **Anais...** IEEE Computer Society, 2008. p.325–332.

APÊNDICES

APÊNDICE A – Árvores de Derivação

Apêndice que contêm as provas de tipos de expressões apresentadas ao longo da dissertação. Conforme explicado na Seção 3.1, uma derivação deve ser lida de baixo para cima, ou seja, utilizando o conceito de árvore. A parte de baixo da derivação é a raiz e a parte superior são as folhas. Cada sentença da derivação é obtida por alguma regra do sistema de tipos. Uma sentença será válida se sempre existir alguma regra que permita expandir a derivação, até chegar em um nó folha.

$$\frac{\Gamma \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] : Shh \textbf{ [prova na árvore 1]} \qquad \Gamma \vdash B[open\ msg.(x : W).P] : Shh \textbf{ [prova na árvore 2]}}{\Gamma, A : Amb[Shh], B : Amb[W], msg : Amb[W], M : W, P : W \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] \mid B[open\ msg.(x : W).P] : Shh}\ \text{(Proc Par)}$$

(a) árvore principal

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash A : Amb[Shh]}\ \text{(Exp } n\text{)}}{\dfrac{\Gamma \vdash out\ A : Cap[W]}{\Gamma \vdash out\ A.in\ B : Cap[W]}\ \text{(Exp } out\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash B : Amb[W]}\ \text{(Exp } n\text{)}}{\Gamma \vdash in\ B : Cap[W]}\ \text{(Exp } in\text{)}}\ \text{(Exp .)} \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : W}\ \text{(Proc Zero)}}{\Gamma \vdash (out\ A.in\ B).\mathbf{0} : W}\ \text{(Proc Ação)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash M : W}\ \text{(Exp } n\text{)}}{\Gamma \vdash \langle M \rangle : W}\ \text{(Proc Saída)}}{\Gamma \vdash (out\ A.in\ B).\mathbf{0} \mid \langle M \rangle : W}\ \text{(Proc Par)}$$

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash A : Amb[Shh]}\ \text{(Exp } n\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash msg : Amb[W]}\ \text{(Exp } n\text{)}}{\Gamma \vdash msg[out\ A.in\ B \mid \langle M \rangle] : Shh}\ \text{(Proc Amb)}}{\Gamma \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] : Shh}\ \text{(Proc Amb)}$$

(b) árvore 1

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash B : Amb[W]}\ \text{(Exp } n\text{)} \quad \dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash msg : Amb[W]}\ \text{(Exp } n\text{)}}{\Gamma \vdash open\ msg : Cap[W]}\ \text{(Exp } open\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma, n : W \vdash P : W}\ \text{(Env } n\text{)}}{\Gamma \vdash (x : W).P : W}\ \text{(Proc Entrada)}}{\Gamma \vdash open\ msg.(x : W).P : W}\ \text{(Proc Ação)}}{\Gamma \vdash B[open\ msg.(x : W).P] : Shh}\ \text{(Proc Amb)}$$

(c) árvore 2

Figura A.1 – Prova que a expressão pacote tipado da página 50 é bem tipada

$$\frac{\Gamma \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] : Shh \textbf{ [prova na árvore 1]} \qquad \Gamma \vdash B[open\ msg.(x : W).P] : Shh \textbf{ [prova na árvore 2]}}{\Gamma, A : Amb[Shh], B : Amb[Shh], msg : Amb[W], M : W, P : W \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] \mid B[open\ msg.(x : W).P] : Shh}\ \text{(Proc Par)}$$

(a) árvore principal

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash A : Amb[Shh]}\ \text{(Exp } n\text{)}}{\Gamma \vdash out\ A : Cap[W]}\ \text{(Exp } out\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash B : Amb[Shh]}\ \text{(Exp } n\text{)}}{\Gamma \vdash in\ B : Cap[W]}\ \text{(Exp } in\text{)}}{\Gamma \vdash out\ A.in\ B : Cap[W]}\ \text{(Exp .)} \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : W}\ \text{(Proc Zero)}}{\Gamma \vdash (out\ A.in\ B).\mathbf{0} : W}\ \text{(Proc Ação)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash M : W}\ \text{(Exp } n\text{)}}{\Gamma \vdash \langle M \rangle : W}\ \text{(Proc Saída)}}{\Gamma \vdash (out\ A.in\ B).\mathbf{0} \mid \langle M \rangle : W}\ \text{(Proc Par)}}{\Gamma \vdash msg[out\ A.in\ B \mid \langle M \rangle] : Shh}\ \text{(Proc Amb)}}{\Gamma \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] : Shh}\ \text{(Proc Amb)}$$

(b) árvore 1

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash B : Amb[Shh]}\ \text{(Exp } n\text{)} \quad \dfrac{\dfrac{\text{ERRO, regra não satisfeita}}{\Gamma \vdash open\ msg : Cap[Shh]}\ \text{(Exp } open\text{)} \quad \dfrac{\text{ERRO, regra não satisfeita}}{\Gamma \vdash (x : W).P : Shh}\ \text{(Proc Entrada)}}{\Gamma \vdash open\ msg.(x : W).P : Shh}\ \text{(Proc Ação)}}{\Gamma \vdash B[open\ msg.(x : W).P] : Shh}\ \text{(Proc Amb)}$$

(c) árvore 2

Figura A.2 – Prova que a expressão pacote tipado da página 50 não é bem tipada se $B : Amb[Shh]$

$$\frac{\Gamma \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] : Shh \textbf{ [prova na árvore 1]} \qquad \Gamma \vdash B[open\ msg.(x : W).P] : {}^{\veebar}Shh \textbf{ [prova na árvore 2]}}{\Gamma, A : Amb^{\bullet}[{}^{\veebar}Shh], B : Amb^{\bullet}[{}^{\frown}W], msg : Amb^{\circ}[{}^{\frown}W], M : W, P : {}^{\frown}W \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] \mid B[open\ msg.(x : W).P] : {}^{\veebar}Shh}\ \text{(Proc Par)}$$

(a) árvore principal

$$\dfrac{\dfrac{\Gamma \vdash A : Amb^{\bullet}[{}^{\veebar}Shh]}{\Gamma \vdash A : Amb^{\bullet}[{}^{\veebar}Shh]}\ \text{(Exp } n\text{)} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash A : Amb^{\bullet}[{}^{\veebar}Shh]}\ \text{(Exp } n\text{)}}{\Gamma \vdash out\ A : Cap[{}^{\frown}W]}\ \text{(Exp } out\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash B : Amb^{\bullet}[{}^{\frown}W]}\ \text{(Exp } n\text{)}}{\Gamma \vdash in\ B : Cap[{}^{\frown}W]}\ \text{(Exp } in\text{)}}{\Gamma \vdash out\ A.in\ B : Cap[{}^{\frown}W]}\ \text{(Exp .)} \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : {}^{\frown}W}\ \text{(Proc Zero)}}{\Gamma \vdash (out\ A.in\ B).\mathbf{0} : {}^{\frown}W}\ \text{(Proc Ação)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash M : W}\ \text{(Exp } n\text{)}}{\Gamma \vdash \langle M \rangle : {}^{\frown}W}\ \text{(Proc Saída)}}{\Gamma \vdash (out\ A.in\ B).\mathbf{0} \mid \langle M \rangle : {}^{\frown}W}\ \text{(Proc Par)}}{\Gamma \vdash msg[out\ A.in\ B \mid \langle M \rangle] : {}^{\veebar}Shh}\ \text{(Proc Amb)}}{\Gamma \vdash A[msg[out\ A.in\ B \mid \langle M \rangle]] : {}^{\veebar}Shh}\ \text{(Proc Amb)}$$

(b) árvore 1

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash B : Amb^{\bullet}[{}^{\frown}W]}\ \text{(Exp } n\text{)} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash msg : Amb^{\circ}[{}^{\frown}W]}\ \text{(Exp } n\text{)}}{\Gamma \vdash open\ msg : Cap[{}^{\frown}W]}\ \text{(Exp } open\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma, n : W \vdash P : {}^{\frown}W}\ \text{(Env } n\text{)}}{\Gamma \vdash (x : W).P : {}^{\frown}W}\ \text{(Proc Entrada)}}{\Gamma \vdash open\ msg.(x : W).P : {}^{\frown}W}\ \text{(Proc Ação)}}{\Gamma \vdash B[open\ msg.(x : W).P] : {}^{\veebar}Shh}\ \text{(Proc Amb)}$$

(c) árvore 2

Figura A.3 – Prova que a expressão pacote tipado da página 51 é bem tipada

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash bob : \textbf{\textit{Amb}}[Shh]}\ (\text{Exp } n) \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash phone : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\ (\text{Exp } n) \quad \Gamma \vdash at(conf)?switchto\langle silent\rangle.\mathbf{0} \mid out.\mathbf{0} : \textbf{\textit{Cap}}[W]\ \textbf{[árvore 1]}}{\Gamma \vdash phone[at(conf)?switchto\langle silent\rangle.\mathbf{0} \mid out.\mathbf{0}] : Shh}\ (\text{Proc Amb})}{\Gamma, bob\colon \textbf{\textit{Amb}}[Shh], phone\colon \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]], silent\colon W, switchto\colon \textbf{\textit{Abs}}[W] \vdash bob[phone[at(conf)?switchto\langle silent\rangle.\mathbf{0} \mid out.\mathbf{0}]] : Shh}\ (\text{Proc Amb})$$

(a) árvore principal

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash silent : W}\ (\text{Exp } n) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash switchto : \textbf{\textit{Abs}}[W]}\ (\text{Exp } n)}{\Gamma \vdash switchto\langle silent\rangle : \textbf{\textit{Cap}}[W]}\ (\text{Proc Cham } \varepsilon) \quad \dfrac{\Gamma \vdash \diamond}{\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Proc Zero})}{\Gamma \vdash switchto\langle silent\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Proc Ação})}{\Gamma, at(conf) : Bool \vdash switchto\langle silent\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Env } k)}{\Gamma \vdash at(conf)?switchto\langle silent\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Proc Contex}) \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash out : \textbf{\textit{Cap}}[W]}\ (\text{Exp } out) \quad \dfrac{\Gamma \vdash \diamond}{\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Proc Zero})}{\Gamma \vdash out.\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Proc Ação})$$

$$\dfrac{\Gamma \vdash at(conf)?switchto\langle silent\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W] \quad \Gamma \vdash out.\mathbf{0} : \textbf{\textit{Cap}}[W]}{\Gamma \vdash at(conf)?switchto\langle silent\rangle.\mathbf{0} \mid out.\mathbf{0} : \textbf{\textit{Cap}}[W]}\ (\text{Proc Par})$$

(b) árvore 1

Figura A.4 – Prova que a expressão bob phone da página 67 é bem tipada

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash proj : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\ (\text{Exp } n) \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma, x : W \vdash P : \textbf{\textit{Cap}}[W]}\ (\text{Env } n) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash pess : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\ (\text{Exp } n)}{\Gamma \vdash pess :: (x : W).P : \textbf{\textit{Cap}}[W]}\ (\text{Proc Entrada } \alpha)}{\Gamma \vdash proj[pess :: (x : W).P] : Shh}\ (\text{Proc Amb}) \quad \Gamma \vdash pess[proj :: \langle M \rangle] : Shh\ \textbf{[árvore 1]}}{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash sala : \textbf{\textit{Amb}}[Shh]}\ (\text{Exp } n) \quad \Gamma \vdash proj[pess :: (x : W).P] \mid pess[proj :: \langle M \rangle] : Shh}{\Gamma, sala\text{: } \textbf{\textit{Amb}}[Shh], proj, pess\text{: } \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]], P\text{: } \textbf{\textit{Cap}}[W], M\text{: } W \vdash sala[proj[pess :: (x : W).P] \mid pess[proj :: \langle M \rangle]] : Shh}\ (\text{Proc Amb})}\ (\text{Proc Par})$$

(a) árvore principal

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash pess : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\ (\text{Exp } n) \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash M : W}\ (\text{Exp } n) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash proj : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\ (\text{Exp } n)}{\Gamma \vdash proj :: \langle M \rangle : \textbf{\textit{Cap}}[W]}\ (\text{Proc Saída } \alpha)}{\Gamma \vdash pess[proj :: \langle M \rangle] : Shh}\ (\text{Proc Amb})$$

(b) árvore 1

Figura A.5 – Prova que a expressão projetor da página 68 é bem tipada

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash AL : W}\,(\text{Exp } n) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash EnviaMsg : \textbf{\textit{Abs}}[W]}\,(\text{Exp } n)}{\Gamma \vdash Hosp \uparrow EnviaMsg\langle AL\rangle : \textbf{\textit{Cap}}[W]}\,(\text{Proc Cham } \alpha) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \textbf{\textit{Cap}}[W]}\,(\text{Proc Zero})}{\Gamma, Perigo : Bool \vdash Hosp \uparrow EnviaMsg\langle AL\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W]}\,(\text{Env } k) \,/\, (\text{Proc Ação})}{\Gamma, EnviaMsg : \textbf{\textit{Abs}}[W], Hosp : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]], AL : W \vdash Perigo?Hosp \uparrow EnviaMsg\langle AL\rangle.\mathbf{0} : \textbf{\textit{Cap}}[W]}\,(\text{Proc Contex})$$

(a) prova da expressão a

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma, M : W \vdash EnviaMsg : \textbf{\textit{Abs}}[W]}\,(\text{Env } n) \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash M : W}\,(\text{Exp } n) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Enf : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\,(\text{Exp } n)}{\Gamma \vdash Enf \downarrow \langle M\rangle : \textbf{\textit{Cap}}[W]}\,(\text{Proc Saída } \alpha) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Pac : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\,(\text{Exp } n)}{\Gamma, EnviaMsg : \textbf{\textit{Abs}}[W], Pac, Enf : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]] \vdash EnviaMsg : \textbf{\textit{Abs}}[W] \triangleright\ Pac \downarrow (M : W).Enf \downarrow \langle M\rangle : \textbf{\textit{Cap}}[W]}\,(\text{Proc Abs } \alpha)$$

(b) prova da expressão b

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \textbf{\textit{Cap}}[W]}\,(\text{Proc Zero})}{\Gamma, M : W \vdash \mathbf{0} : \textbf{\textit{Cap}}[W]}\,(\text{Env } n) \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Hosp : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\,(\text{Exp } n)}{\Gamma, Hosp : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]] \vdash Hosp \uparrow (M : W).\mathbf{0} : \textbf{\textit{Cap}}[W]}\,(\text{Proc Entrada } \alpha)$$

(c) prova da expressão c

Figura A.6 – Prova que as Expressões 1 da página 71 são bem tipadas

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \textbf{\textit{Cap}}[W]}\,\text{(Proc Zero)}}{\Gamma, M : W \vdash \mathbf{0} : \textbf{\textit{Cap}}[W]}\,\text{(Env } n\text{)} \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Hosp : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\,\text{(Exp } n\text{)}}{\Gamma \vdash Hosp \uparrow (M : W).\mathbf{0} : \textbf{\textit{Cap}}[W]}\,\text{(Proc Entrada } \alpha\text{)} \quad \Gamma \vdash in\ Quar.Quar \uparrow \langle Aut \rangle : \textbf{\textit{Cap}}[W]\textbf{[árvore 1]}}{\Gamma, Hosp, Quar : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]], Aut : W \vdash Hosp \uparrow (M : W).\mathbf{0} : \textbf{\textit{Cap}}[W] \mid in\ Quar.Quar \uparrow \langle Aut \rangle : \textbf{\textit{Cap}}[W]}\,\text{(Proc Par)}$$

(a) árvore principal

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Quar : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\,\text{(Exp } n\text{)}}{\Gamma \vdash in\ Quar : \textbf{\textit{Cap}}[W]}\,\text{(Exp } in\text{)} \quad \dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Aut : W}\,\text{(Exp } n\text{)} \quad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash Quar : \textbf{\textit{Amb}}[\textbf{\textit{Cap}}[W]]}\,\text{(Exp } n\text{)}}{\Gamma \vdash Quar \uparrow \langle Aut \rangle : \textbf{\textit{Cap}}[W]}\,\text{(Proc Saída } \alpha\text{)}}{\Gamma \vdash in\ Quar.Quar \uparrow \langle Aut \rangle : \textbf{\textit{Cap}}[W]}\,\text{(Proc Ação)}$$

(b) árvore 1

Figura A.7 – Prova que a Expressão Pe da página 72 é bem tipada