UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FELIPE DE OLIVEIRA TANUS

Formalização de ACCE no provador de teoremas Coq

Trabalho de Conclusão

Prof. Dr. Alvaro Moreira
Orientador

Porto Alegre, Janeiro de 2013

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

de Oliveira Tanus, Felipe

Formalização de ACCE no provador de teoremas Coq / Felipe de Oliveira Tanus. – Porto Alegre: PPGC da UFRGS, 2013.

43 f.: il.

Trabalho de Conclusão – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2013. Orientador: Alvaro Moreira.

1. LLVM. 2. VeLLVM. 3. Tolerância a Falhas. 4. ACCE. 5. Coq. I. Moreira, Alvaro. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

AGRADECIMENTOS

Agradeço a minha família e namorada, pelo apoio e companhia. Ao Álvaro Moreira pela ótima orientação durante o trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

LLVM	Low Level Virtual Machine
LLVM IR	LLVM Intermediate Language
VeLLVM	Verified LLVM
SSA	Static Single Assignment form
CPS	Continuation-Passing style
CFE	Control Flow Error
ACCE	Automatic Correction of Control-flow Errors

LISTA DE FIGURAS

RESUMO

Este trabalho descreve uma implementação da transformação ACCE em Coq. A técnica visa permitir que um programa detecte e corrija automaticamente erros de controle de fluxo causados por *soft errors*. A transformação é aplicada na linguagem intermediária LLVM IR. Para isso, é utilizado o VeLLVM, que disponibiliza uma implementação de uma semântica formal para a linguagem LLVM IR. Esse é o primeiro passo para a construção de provas formais sobre a técnica ACCE.

Palavras-chave: LLVM, VeLLVM, Tolerância a Falhas, ACCE, Coq.

Formalization of ACCE technique on Coq formal proof management system

ABSTRACT

This work describes an implementation in Coq of ACCE. This technique aims to allow a software to detect and automatically fix control flux errors, generaly caused by soft errors. This transformation is applied on the LLVM Intermediate language. To accomplish this, the LLVM IR formal semantic presented by the VeLLVM is used. This is the first step to write proofs about the ACCE technique.

Keywords: LLVM, VeLLVM, Fault Tolerance, ACCE, Coq.

1 INTRODUÇÃO

Graças à diminuição gradual do tamanho dos transistores, uma das principais barreiras da computação é a dissipação de potência das máquinas. O próximo grande desafio nessa linha é tratar os *soft errors*, pois são mais comuns à medida que o tamanho dos transistores diminui (MUKHERJEE, 2008). Um estudo mais detalhado desses erros antes mesmo de o problema se tornar uma barreira definitiva é importante para que um avanço rápido da tecnologia ocorra.

Soft errors são erros naturais, difíceis de serem evitados. Todo computador está sujeito a esses erros. Eles são muitas vezes causados pela incidência de radiação nos componentes eletrônicos, sendo capazes de mudar os valores lógicos de alguns bits ou até mesmo a próxima instrução a ser executada. Isso pode trazer consequências drásticas para um sistema de computação.

Diversas técnicas em nível de hardware e software existem para detectar esses erros. Este trabalho traz uma abordagem formal para a transformação ACCE (*Automatic Correction of Control-flow Errors*), técnica de software que visa a evitar e corrigir os *soft errors* que modificam a parte do programa que está sendo executada em um dado momento, por exemplo um *bitflip* no *program counter* de uma máquina. O principal objetivo do trabalho é formalizar a técnica e, assim, possibilitar a construção de provas sobre ela.

Para isso, iremos utilizar LLVM (*Low Level Virtual Machine*), um conjunto completo de ferramentas para a compilação de código, inclusive contando com a LLVM IR (LLVM *Intermediate Language*), uma linguagem intermediária. O LLVM se prova bastante útil para uma abordagem formal para o estudo de *soft errors* pois a LLVM IR é bem definida e é de um nível de abstração baixo, isso é, para a maioria das máquinas atuais, cada instrução do LLVM IR é traduzida para uma instrução de processador.

Felizmente, foi lançado recentemente um trabalho que define uma semântica operacional para a linguagem, o que facilita o trabalho de especificação da transformação, chamado de VeLLVM (*Verified* LLVM) (ZHAO et al., 2012). O presente trabalho se propõe a implementar a técnica ACCE no *framework* oferecido pelo projeto LLVM, a fim de dar o primeiro passo para uma análise mais rigorosa sobre a técnica.

1.1 Contexto e Motivação

Soft errors são erros extremamente difíceis de lidar, pois destroem as regras esperadas do sistema de computação como nós o conhecemos. Para criar programas confiáveis levando em consideração *soft errors* não podemos confiar que ao ler uma variável gravada encontraremos o valor esperado, ou então que o computador irá seguir as instruções sequencialmente. As regras do jogo a que estamos acostumados mudam drasticamente.

Uma maneira de atacar as possibilidades numerosas que se leva em consideração ao tratar *soft errors* é raciocinar formalmente sobre o problema. Uma boa linguagem para raciocinar sobre *soft errors* é LLVM IR (*Low Level Virtual Machine Intermediate Representation*), pois ela é baixo nível o suficiente para tornar mais evidente os efeitos sobre o fluxo de controle de *soft errors* e ao mesmo tempo possui uma semântica formal definida no provador de teoremas Coq (ZHAO et al., 2012). Reutilizar esse trabalho e raciocinar sobre a técnica apresentada é uma maneira de chegar a conclusões confiáveis sobre ela.

1.2 Objetivos

Este trabalho tem como objetivo apresentar formalmente a transformação ACCE, que lida com erros de controle de fluxo. Existe uma série de raciocínios formais que devem ser feitos em uma transformação para que ela possa ser utilizada seguramente. Essa formalização é o primeiro passo para que se possa raciocinar sobre a técnica.

Um dos objetivos deste trabalho é abrir as portas para um estudo formal detalhado da transformação ACCE. Outro objetivo é trazer um relato detalhado do *framework* VeLLVM, uma vez que ele pode ser utilizado para raciocinar sobre outras transformações.

Dentre os raciocínios que são interessantes de serem desenvolvidos para a técnica ACCE, temos a prova de que a semântica de qualquer programa não é alterada ao aplicar a transformação. Isso é necessário porque uma transformação que evite erros de fluxos, mas que mude o resultado de um programa, não é útil. Outro raciocínio importante consiste em definir um modelo formal de falha e verificar se para esse modelo a transformação ACCE é bem sucedida, ou seja, se o ACCE cumpre seu propósito de detectar e corrigir determinados erros de fluxo de controle.

Acima de tudo, este trabalho visa a dar um passo inicial no estudo formal da transformação ACCE e trazer, à comunidade acadêmica, uma experiência sobre raciocínio formal de transformações utilizando o *framework* VeLLVM.

1.3 Trabalhos Relacionados

A técnica ACCE para detecção e correção de determinados erros de fluxo de controle causados por *soft errors* foi definida no artigo (VEMU; GURUMURTHY; ABRAHAM, 2007). O artigo relata resultados experimentais feitos com a implementação da técnica para o GCC.

A dissertação de mestrado de Rafael Baldiatti Parizi (PARIZI, 2012) descreve a implementação da técnica ACCE como um passo de transformação do *framework* de compilação LLVM aplicado sobre programas na linguagem intermediária LLVM-IR. Essa dissertação detalha os resultados de diversos experimentos que mostram a eficácia na detecção e correção e também como ela é afetada quando outras transformações do LLVM são aplicadas. Nenhum desses trabalhos, porém, trata da correção da técnica, ou seja, nenhum dos trabalhos prova formalmente se a transformação ACCE mantém a semântica original do programa e se ela de fato detecta e corrige todas as falhas a que se propõe detectar e corrigir.

Recentemente a semântica formal da linguagem intermediária LLVM-IR foi formalizada no provador de teoremas Coq (ZHAO et al., 2012). Essa formalização, juntamente com a implementação de ACCE em LLVM, são utilizados como ponto de partida para a formalização da transformação ACCE em Coq definida neste trabalho de conclusão.

1.4 Organização do texto

O capítulo dois tem enfoque em tolerância a falhas, apresentando conceitos básicos para que o leitor não tenha dificuldades em prosseguir a leitura. No mesmo capítulo, são definidos *soft errors* e erros de controle de fluxo. Em seguida, a técnica ACCE é apresentada em detalhes.

No terceiro capítulo, o enfoque passa para LLVM e VeLLVM. É explicado o básico sobre LLVM e LLVM IR, que é a linguagem intermediária do compilador LLVM definida pelo VeLLVM. A seguir, é introduzido o VeLLVM, *framework* para raciocínio em cima da LLVM IR, é dada uma explicação da semântica definida. Em seguida é apresentada a organização do código disponibilizado pelo VeLLVM, produzida neste trabalho.

O quarto capítulo relata o início da implementação do ACCE no framework VeLLVM, começando com a preparação do ambiente. Em seguida temos as modificações necessárias no *framework* para que o trabalho pudesse ser completado. Então é apresentada a implementação do ACCE, passando pelas etapas de cálculo de assinaturas e adição de código.

Na conclusão é feita uma discussão sobre quais seriam os próximos passos na realização de provas de equivalência semântica, e também da efetividade da técnica quanto à detecção e correção de *soft errors*.

2 TOLERÂNCIA A FALHAS

2.1 Conceitos Básicos

Todo conteúdo deste capítulo é fortemente baseado no artigo de Avizienis (AVIZIENIS et al., 2004), que reúne uma grande parte das definições básicas feitas na área de tolerância a falhas.

2.1.1 Sistema, Função do Sistema, Comportamento, Estrutura e Serviço

Sistema é uma entidade que interage com outras entidades, isso é, outros sistemas, incluindo hardware, software, humanos e o mundo físico. Um sistema tem uma ou mais funções. Uma função é o que se espera que um sistema faça, descrito na especificação funcional do sistema.

Para implementar uma função, um sistema tem um comportamento, que é descrito como uma sequencia de estados. O estado total de um sistema é o conjunto dos seguintes estados: computação, comunicação, interconexão e condição.

A estrutura de um sistema é o que possibilita a ele ter um determinado comportamento. De um ponto de vista estrutural, o sistema é composto de um conjunto de componentes unidos que interagem. Cada um desses componentes é um sistema. A recursão para quando chegamos em um componente considerado atômico. Qualquer estrutura interna que não pode ser avaliada ou que não é interessante pode ser ignorada. Consequentemente, o estado total de um sistema é o conjunto de estados externos de seus componentes atômicos.

Um serviço é um comportamento de um sistema como visto pelos seus usuários. Um usuário é outro sistema que recebe serviços de um provedor. A parte de um sistema onde o serviço é entregue se chama interface de serviço. A parte dos estados totais do sistema servidor é conhecida como interface externa, e a outra parte é sua interface interna. Um serviço correto é entregue quando o serviço implementa a função do sistema.

2.1.2 Tipos de ameaças: Defeito, Erro e Falha

Um *defeito* de serviço é um evento que ocorre quando um sistema para de entregar um serviço correto. Um serviço é defeituoso porque não cumpre com a especificação

de função ou porque a especificação da função não descreve adequadamente a função do sistema.

O período de entrega de um serviço incorreto (isso é, um serviço que não é correto) é nomeado serviço interrompido. A transição do serviço incorreto para o serviço correto, que é a transição contrária de um defeito de serviço, é conhecida como recuperação do serviço.

Já que um serviço é a sequência de seus estados externos, um defeito de serviço significa que ao menos um estado externo do sistema deixou de ser o estado correto do serviço. Quando um estado deixa de ser o estado esperado para gerar um estado correto do serviço, occore um *erro*. O culpado por ocorrer esse erro é chamada de *falha*.

Falhas podem ser externas ou internas a um sistema. Uma vulnerabilidade é a uma falha interna que possibilita a uma falha externa prejudicar o sistema. Para haver uma falha externa, uma vulnerabilidade é necessária.

Normalmente, uma falha causa um erro em um estado interno de um serviço, e o estado externo não é imediatamente afetado. Por essa razão, a definição de um erro é a parte do estado total de um sistema que pode levar a um defeito de serviço. É importante notar que muitos erros não alcançam os estados externos de um sistema e causam um defeito. Uma falha é dita ativa quando causa um erro. Caso contrário, é dita dormente.

2.2 *Soft Errors*

Alguns erros podem ocorrer quando componentes eletrônicos são expostos a partículas alfa energizadas, presentes na radiação. Uma partícula alfa, independentemente de sua origem, pode penetrar até 25 micrômetros. Ao penetrar em um componente de hardware, perturbando os elétrons o suficiente, pode ocorrer um *bit flip*, ou seja, a mudança de estado de um bit. Esses erros são conhecido como *Soft Errors* (MUKHERJEE, 2008).

Outra fonte conhecida de *Soft Errors* são os raios cósmicos, gerados pelo sol. Apesar de a partícula primária dos raios cósmicos geralmente não atingir a superfície da terra, ela cria uma chuva de partículas secundárias. Na superfície da terra, aproximadamente 95% das partículas capazes de gerar *soft errors* são nêutrons energizados com o restante composto de prótons e píons. Colocar o computador em uma caverna reduz a taxa de erros causados por raios cósmicos a um nível irrelevante (TANG, 1996).

Entre as fontes de *Soft Errors* podemos citar os Nêutrons. Ao chegar no equilíbrio térmico podem gerar reações de captura neutrônica com os circuitos. Essas reações produzem partículas alfa, núcleos de ^{7}Li e raios gama (WEN et al., 2010). Há ainda outras fontes de *Soft Errors*, como, por exemplo, ruídos aleatórios ou problemas de integridade de sinal como diafonia (*crosstalk*), porém acredita-se que essas fontes de erros tenham uma pequena contribuição em relação às outras citadas (BHATTACHARYA; RANGANATHAN, 2009).

A computação, desde sua criação, vem enfrentado uma série de desafios. Há algum tempo, o principal problema da computação era o acesso a memórias dinâmicas. Houve então um avanço na tecnologia que permitiu que os microprocessadores lidassem com latências maiores de memória. Recentemente, a potência dos processadores acabou influenciando na evolução dos processadores, fazendo com que os fabricantes de microprocessadores prestassem maior atenção na dissipação da potência. Nessa série de desafios, as falhas transientes geradas por partículas alfa são as próximas a serem enfrentadas (MUKHERJEE, 2008).

A primeira vez que se tem notícia de *soft errors* na terra foi em 1987, reportado pela Intel. Na época, a Intel havia sido contratada pela AT&T para fabricar componentes eletrônicos para converter seus sistemas de relé para circuitos integrados. A entrega no tempo combinado não foi possível por causa desses *soft errors*. A Intel conseguiu encontrar o problema no empacotamento de seus circuitos, que são feitos com material radioativo. Até hoje o material do empacotamento dos circuitos é feito com material fracamente radioativo, porém hoje em dia é feito um controle de qualidade por empresas como IROCTech (IROCTECH, 2012), normalmente mantendo o nível de emissão abaixo de 0.001 cph/cm^2. O nível de emissão permitido nos empacotamento de circuitos baixa conforme o os resistores ficam menores (MUKHERJEE, 2008).

2.3 ACCE

ACCE (*Automatic correction of control-flow errors*) foi a primeira técnica de controle de fluxo com correção automática a ser publicada. É uma técnica que visa à detecção e correção de erros. A sua principal diferença em relação a outras técnicas de correção de *soft errors* é não gravar um conjunto de estados do processador a cada momento em que o fluxo é testado, conseguindo assim uma performance melhor, necessária para sistemas em tempo real. Para atingir esse objetivo, é utilizada redundância de código para o programa. No trabalho original (VEMU; GURUMURTHY; ABRAHAM, 2007), o ACCE foi implementado como uma modificação do GCC, e testes de medição de performance apontam que o *overhead* é muito baixo. Ao injetar falhas utilizando os programas de *benchmark SPEC* e *MiBench* em programas com a modificação, foi constatado que a saída produzida pelos programas compilados com ACCE continua a mesma com uma probabilidade de mais 90%, com um *overhead* de algumas centenas de instruções (VEMU; GURUMURTHY; ABRAHAM, 2007).

É importante notar que no artigo original (VEMU; GURUMURTHY; ABRAHAM, 2007), não há nenhuma prova de que a técnica ACCE mantém a semântica original do programa, ou que a técnica é efetiva. Para comprovar o funcionamento, apenas a intuição das provas são apresentadas, e a transformação implementada para GCC é testada utilizando *frameworks* de teste, injetando falhas artificiais.

As seções a seguir foram totalmente baseadas no artigo original da técnica ACCE (VEMU; GURUMURTHY; ABRAHAM, 2007). As referências serão omitidas para uma melhor legibilidade.

2.3.1 Nomenclatura

Para explicar o funcionamento do ACCE, a seguinte nomenclatura é utilizada:

Grafo do programa é um grafo $P = (V, E)$ onde V são os nodos N_1, N_2, ..., N_n que equivalem a a blocos básicos do LLVM, conforme explicado na seção 3.1.1 e E são as arestas $N_i \rightarrow N_j$, representando um possível desvio de fluxo do bloco N_i para o bloco N_j.

O **tipo de um nodo**, representado por $NT(N)$, pode ser A ou X. Um nodo é do tipo A se ele possuir múltiplos predecessores e ao menos um desses predecessores possui mais de um sucessor. Caso contrário, é do tipo X.

Existe uma **variável de assinatura** S que é local para cada função, sendo utilizada para monitorar a execução do programa. Cada nodo possui uma **assinatura de entrada** NS e uma **assinatura de saída** NES, que são os valores esperados de S ao entrar em um nodo e ao sair de um nodo, respectivamente, para a execução ser considerada correta. Essas assinaturas são definidas para cada nodo em tempo de compilação. Cada nodo também possui **parâmetros** $d1$ e $d2$, que são utilizados para atualizar S, baseados no NS e no NES. Esse sistema é utilizado para identificar os CFEs (*Control Flow Error*).

Cada função tem um identificador único, definido em tempo de compilação. A **função identidade** de uma função do programa é definida como $FID(f)$, e retorna o identificador único fid de uma função f. Esse identificador único é posteriormente utilizado para a correção automática.

2.3.2 Base do funcionamento

Para cada nodo, NS e NES são atribuídos estaticamente, juntamente com um código responsável por testar o valor de S. Em tempo de execução, S é testada contra o seu valor esperado Se, baseado no NS e NES, e atualizado no início e no fim de cada nodo que é executado. Um erro entre o valor atual e o esperado de S sinaliza que um CFE ocorreu. O fluxo do programa é desviado para uma função global que corrige o erro, que também é adicionada estaticamente. Quando o programa é desviado para essa função, ela detecta a função que foi executada imediatamente antes do CFE e transfere o programa novamente para aquela função.

As próximas seções explicam em detalhes os mecanismos de detecção e correção de erros.

2.3.3 Detecção de erro

Para a detecção de erros duas instruções são adicionadas estaticamente no início de um nodo, formando o cabeçalho, e outras duas no final de um nodo, formando o rodapé. O cabeçalho do bloco depende do tipo do nodo, enquanto o rodapé é igual para todos os nodos. Como já mencionado, NS e NES são determinados estaticamente. A execução da primeira instrução do cabeçalho testa se houve desvio ilegal. Então $d1(N)$ é atribuído de forma que depois da execução do cabeçalho o valor de S dentro do nodo seja igual ao

valor esperado $NS(N)$. A primeira instrução do rodapé testa novamente por um desvio ilegal diretamente para uma instrução de dentro do nodo N. A segunda instrução atribui $d2(N)$ de tal forma que após a execução do rodapé, o valor de S seja igual a $NES(S)$. Em ambos os casos, caso a comparação falhe, o programa é desviado para a função tratadora de errors *err* definida em 2.3.4.

A figura 2.1 mostra como as variáveis de assinatura são atualizadas de forma a manter os requerimentos da técnica. Em 2.1(a) é mostrado o formato de nodo que será usado no resto da figura. O texto entre colchetes informa, em linguagem natural, a situação esperada na execução normal de um programa. Em seguida, primeiramente será expicado como obter NS e NES para os casos 2.1(b) e (c), e em seguida como obter $d1$ e $d2$ para ambos os casos.

A figura 2.1(b) mostra como atribuir as assinaturas para os nodos do tipo X. Todos os predecessores de um nodo do tipo X devem ter o mesmo valor para NES, e esse valor deve ser único. Na figura, podemos notar essa restrição com $NES(N1) = NES(N2)$. $NS(N3)$ deve ser atribuído com um valor único (no caso, 1011). O valor de $d2$ para os nodos $N1$ e $N2$ é calculado de acordo com o valor de seu próprio NES e de $NS(N3)$. Dessa forma, podemos manter o valor de $NS(N3)$ o mesmo, independentemente se $N1$ ou $N2$ são executados, pois o valor de S em ambas as transições é o mesmo. Como os valores atribuídos são únicos, qualquer CFE resulta em uma diferença entre o valor atual e esperado.

Não podemos tratar os nodos do tipo A do mesmo jeito, pois isso permitiria desvios ilegais. A figura 2.1(c) mostra mostra um caso com um nodo de tipo A, que seria o N3. Se utilizássemos o mesmo método para nodos do tipo X, teriamos $NES(N1) = NES(N2)$. Isso faria com que um desvio de $N1$ para $N4$, por exemplo, não fosse detectado.

Podemos evitar esse problema mascarando a diferença de $NES(N1)$ e $NES(N2)$ no início do de $N3$. ACCE, no entanto, introduz uma técnica de detecção que faz essa atualização com apenas uma instrução AND, reduzindo o *overhead*. Porém, o uso da instrução AND pode levar a alguns problemas de mascaramento. Esses problemas podem ser evitados se tomarmos as precauções do parágrafo abaixo.

Devemos dividir os bits das assinaturas NS e NES em duas partes: superior (respectivamente NS_u e NES_u) e inferior (respectivamente NS_l e NES_l). Os valores de NS_u de nodos do tipo A e NES_u de seus pais devem ser o mesmo único valor. Na figura 2.1(c), temos $NS_u(N3) = NES_u(N1) = NES_u(N2)$. Já as partes inferiores devem ser diferentes, de modo que as posições dos bits 0 do NES_l dos predecessores do bloco tipo A sejam subconjuntos próprios das posições dos bits zero do NS_l do bloco tipo A. No exemplo da figura 2.1(c), temos os valores de NES_l dos nodos $N1$ e $N2$ 01 e 10, respectivamente, enquanto $NS_l(N3)$ é 00. Os subconjuntos, nesse caso, ficam 1 para o $N1$, 2 para o $N2$ e 1,2 para o $N3$ (VEMU; GURUMURTHY; ABRAHAM, 2007).

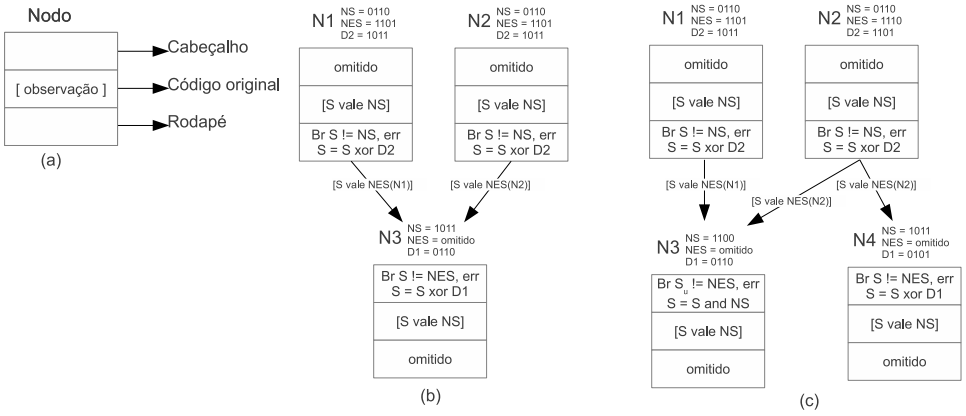

Figura 2.1: Detecção de CFE

2.3.4 Correção automática

Para a técnica corrigir os CFEs encontrados é necessário inserir uma quantidade significativa de código estaticamente, porém a maior parte não é executado quando o programa segue o fluxo esperado .

Como explicado em 2.3.1, cada função possui um identificador único. Esse identificador é retornado pela função identidade $FID(f)$ para uma dada função f. Uma possível tática para definir esse identificador único consiste em definir uma ordem para a lista de funções presentes no código fonte e atribuir a sua posição como identificador único.

No início de cada função e imediatamente após qualquer chamada de função, é inserida uma atribuição $F = FID(f)$, onde F é uma variável global que guarda a função corrente. Note que esse código sempre é executado, independente se houve um CFE ou não. Essa atribuição encontra-se entre o cabeçalho inserido pela etapa de detecção de erro e o código, então sempre há um teste de assinatura imediatamente antes da atribuição.

Ainda para cada função do programa é necessário inserir um pedaço de código, denominado *function error handler* (FEH). Esse código é inserido para lidar com os CFEs dentro da função corrente. Ele só é executado se uma detecção de erro ocorre (sendo então a função *err* presente na figura 2.1), ou então quando ocorre um desvio ilegal diretamente para dentro desse código. O bloco de código 2.1 mostra as instruções a serem adicionadas em pseudocódigo. Considere $Lstarti$ e $Lendi$ *labels* para as instruções que precedem o cabeçalho e o rodapé, respectivamente, adicionados na etapa de detecção de erro, do nodo Ni.

```
1  i1:
2      br F != FID(f), global_error_handler
3  i2:
4      err_flag = 0
5  i3:
6      num_err = num_err +1
7      br num_err > thresh, exit
8  i4:
9      For each node Ni in this function,
10         (a) br S == NS(Ni), Lstarti
11         (a) br S == NES(Ni), Lendi
12 i5:
13     jmp function_error_handler
```

Bloco de código 2.1: Pseudocódigo para FEH

Além da FEH, que é adicionada uma vez para cada função, devemos adicionar uma nova função chamada *global error handler* (GEH). Esse código trata erros de desvios que vão para fora da função corrente. Ele é chamado dentro do FEH, quando é detectado que o desvio é de fora, e quando um desvio ilegal faz com que o código pare no meio dele. O bloco de código 2.2 mostra um pseudocódigo para GEH.

```
1  j1:
2      err_flag = 1
3  j2:
4      For each function f in the program,
5          br F == FID(f), jmp f
6  j3:
7      num_err = num_err + 1
8      br num_err > thresh, exit
9  j4:
10     jmp global_error_handler
```

Bloco de código 2.2: Pseudocódigo para o GEH

A figura 2.2 mostra um exemplo de correção de erro na prática. Um CFE ocorre do nodo $N5$ na função $f1$ para o nodo $N9$ na função $f2$. A partir desse ponto, cada passo apresentado a seguir pode ser conferido no número circulado de cada figura 2.2.

1. O teste de assinatura no final do nodo $N9$ detecta que o valor de S, 0001, é diferente do valor esperado, 11011, e transfere o controle para $f2_err$ (que representa a FEH da função $f2$.

2. A primeira instrução em $f2\_err$ detecta que o valor de F é diferente de $FID(f2)$, e por isso esse CFE trata-se de global, não local na função. A execução é desviada então para o GEH.

3. A primeira instrução em $error\_handler$ atribui 1 a err_flag indicando que o CFE precisa ser tratado. Começa então a procurar pela função a qual o valor de FID é igual ao valor corrente de F. Ele percebe que $f1$ tem esse valor, e transfere o controle para $f1$.

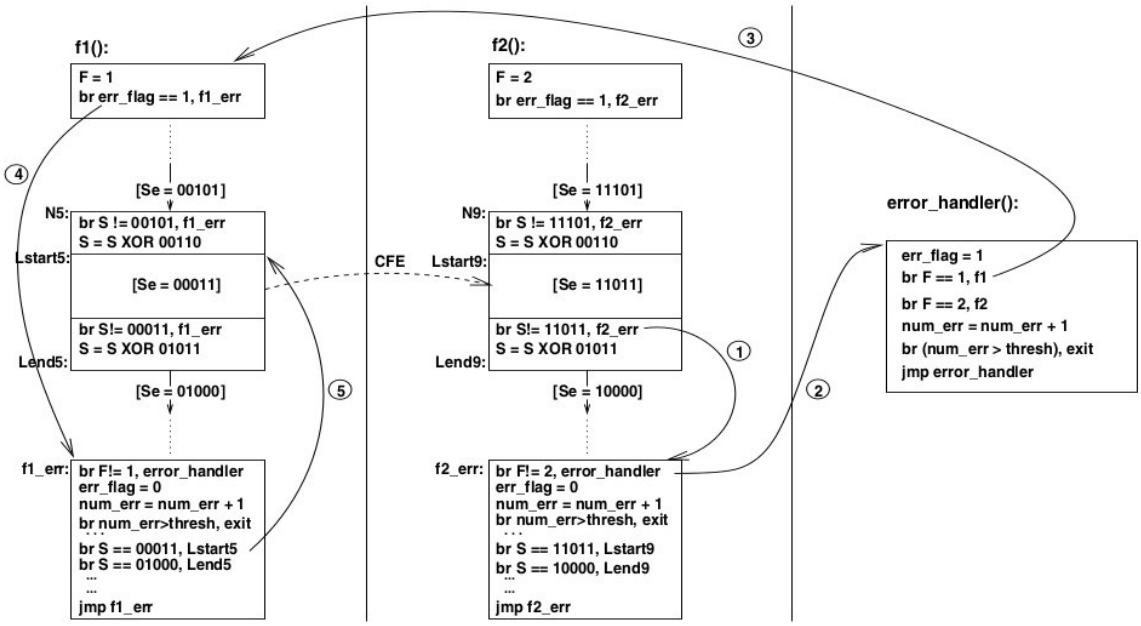

Figura 2.2: Correção de CFEs (VEMU; GURUMURTHY; ABRAHAM, 2007)

4. Em $f1$, um teste do valor de err_flag revela que existem um CFE em tratamento, então o controle é transferido para o FEH da função local, $f1\_err$.

5. A primeira instrução em $f1\_err$ detecta que o valor de F corresponde ao valor do FID da própria função, e então o controle é mantido por $F1\_err$. O Valor 0 é então atribuido a err_flag, significando que o CFE é local. Em seguida detecta que $S = NS(N5)$. O controle então é transferido para o $Lstart5$.

Após a execução dos passos, o programa segue novamente a partir de $N5$, o bloco em que foi interrompido.

Se por algum motivo não houver o valor de S ou de F nos testes da FEH e da GEH, é utilizado um contador que serve para acabar o programa graciosamente através da chamada $exit$, ao invés de deixar um *loop* infinito rodando. O número máximo de entradas em FEH ou GEH é tipicamente 3.

2.3.5 Falhas detectadas pelo hardware

A correção de falhas descrita em 2.3.4 leva em consideração apenas erros de controle de fluxo detectados dentro do programa. Entretanto, a maior parte dos CFEs é detectada pelo *hardware* ou pelo sistema operacional. Esses erros acontecem quando o programa tenta executar instruções de um endereço inválido ou quando tenta executar uma instrução inválida, por exemplo. Esse erro é bem comum se o fluxo do programa é desviado para o meio de uma instrução válida (VEMU; GURUMURTHY; ABRAHAM, 2007).

Esse tipo de erro deve ser tratado transferindo o controle para a função GEH. Em sistemas baseados em Unix, ACCE implementa essa recuperação utilizando o sistema

de *inter-process communication* (IPC) de sinais. Em outros sistemas operacionais que não dispõem desse recurso, a rotina de recuperação de interrupções de hardware deve ser modificada diretamente, e qualquer memória desocupada deve ser preenchida com instruções de *jump* para a função GEH (VEMU; GURUMURTHY; ABRAHAM, 2007).

3 LLVM E VELLVM

3.1 LLVM

O LLVM (*Low Level Virtual Machine*) é um projeto que dispõe de um compilador e todas as ferramentas normalmente disponíveis em uma *toolchain*. Apesar do nome, tem pouco a ver com máquinas virtuais tradicionais, além de prover bibliotecas úteis que podem ser utilizadas para programá-las. Todas essas ferramentas são construídas para lidar com uma forma bem especificada de representação de código, conhecida como *LLVM Intermediate Representation* (LLVM IR). Conta também com uma interface para adicionar facilmente outras linguagens e transformações/otimizações (TEAM, 2012). A imagem 3.1 mostra o funcionamento básico da infra-estrutura do LLVM.

Ao compilar um programa em uma linguagem arbitrária utilizando LLVM, ela é transposta primeiramente para LLVM IR, em seguida para linguagem de montagem. Existe também a possibilidade de compilar um programa para LLVM bytecode, que nada mais é do que uma forma binária compacta do LLVM IR. Nesse estado, o bytecode pode ser interpretado sem necessidade de ser compilado utilizando a ferramenta "lli". Entre as diversas ferramentas disponibilizadas pelo projeto LLVM, podemos citar o compilador clang, o montador llvm-as, o desmontador llvm-dis e o ligador llvm-ld.

O compilador utiliza a estratégia ***static single assignment* (SSA)**, que é uma forma de organizar a linguagem intermediária de modo que cada variável seja atribuída exatamente uma vez. Essa forma é geralmente utilizada para a compilação de linguagens funcionais, como Scheme, ML e Haskell. Ela também é utilizada em muitos outros compiladores como GCC, Jikes e MLton (HUA; XU; GAO, 2010). Como a transformação ACCE in-

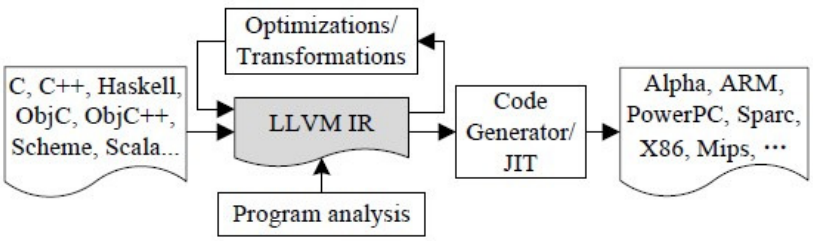

Figura 3.1: Infra-estrutura do LLVM (ZHAO et al., 2012)

sere código no sistema, devemos garantir que ela siga essa organização a fim de gerar programas válidos.

3.1.1 LLVM IR

A linguagem intermediária LLVM IR é uma linguagem de baixo nível cujo propósito é ser compilada na linguagem de montagem de um processador. Ela tem como característica ser muito bem definida e documentada, e é utilizada frequentemente no meio acadêmico como ferramenta para pesquisas por esse motivo.

Para entender como funciona a transformação ACCE, é necessário ter alguma base da sintaxe da LLVM IR. Segue um trecho simples de código LLVM IR, que irá ser descrito em detalhes a seguir:

```
1 define i32 @my_func(i32 %x) {
2 entry:
3     %four = load i32 @four, align 8
4     %five = load i32 @five, align 8
5     %add_result = add i32 %four, %x
6     %icmp_result = icmp eq i32 %add_result, %five
7     ret i32 %x
8 }
```

Bloco de código 3.1: Exemplo de LLVM IR

As linhas 1 a 8 definem uma função com o nome my_func, que recebe uma variável inteira de 32 bits referenciada por x e retorna uma variável de mesmo tipo e tamanho. É interessante notar que antes de uma variável ser referenciada, sempre devemos colocar também o seu tipo e tamanho, por exemplo ao definir a variável de entrada da função. No LLVM IR, os identificadores globais, utilizados para nomes de funções ou variáveis, são precedidos por @. Os identificadores locais, utilizados para nome de variáveis, são precedidos por %.

Das linhas 2 a 7 é definido um bloco básico. Um bloco básico é uma label, que pode ser explícita, como nesse caso, ou implícita, seguida de um conjunto de instruções e um finalizador. O finalizador é uma instrução do tipo branch, condicional ou não, um retorno de função ou ainda a instrução *unreachable*, que indica que a execução do programa nunca deve executar o terminador.

Na segunda linha, há uma *label*. *Labels* são referenciadas por instruções de *branch*, que podem ser condicionais ou incondicionais. A definição de uma *label* é sucedida de dois pontos, porém ao fazer referência a ela, é necessário preceder por %, assim como uma variável local.

As linhas 3 e 4 apresentam o carregamento de duas variáveis globais, @four e @five, para variáveis locais. Devemos especificar o alinhamento dessas variáveis como o último parâmetro da chamada *load*, assim como de muitas outras.

Na linha 5, a instrução *add* é utilizada para somar as duas variáves locais, guardando o resultado em uma variável local.

A linha 6 mostra a utilização da comparação de inteiros. A comparação necessita de uma condição. No exemplo, *eq* vem de *equal* e é a condição de igualdade.

Na linha 7, é chamada a instrução de retorno, retornando a própria variável de entrada.

3.2 Verified LLVM

O *Verified LLVM* (VeLLVM) é um *Framework* para raciocínio sobre programas expressos em LLVM IR e transformações sobre essa linguagem definido no provador de teoremas Coq (ZHAO et al., 2012). Ele apresenta duas semânticas formais, uma na forma de *single-step* e outra na forma de *big-step*. Também inclui o sistemas de tipos dessa linguagem, e uma definição de memória baseada no CompCert (LEROY, 2009).

VeLLVM provê também um interpretador capaz de rodar um LLVM *bytecode*. O interpretador, além de executar o *bytecode* exatamente como a ferramenta lli faria, também gera um traço da computação executada.

Para demonstrar a utilização do *framework* na prática, os autores formalizam e verificam a transformação *SoftBound*, que visa a proteger um programa contra violação de memória (NAGARAKATTE et al., 2009).

3.2.1 Restrições do VeLLVM

Infelizmente, o *framework* cobre apenas uma parte da LLVM IR. O VeLLVM não suporta funções que são usadas exclusivamente para estender o LLVM ou funções bem conhecidas de bibliotecas externas, como da biblioteca padrão do C, por exemplo. Além disso as funções, variáveis e parâmetros nunca podem ser decoradas com atributos que denotam o tipo da ligação, convenções de chamada de função, entre outras possibilidades disponibilizadas pelo LLVM IR. Também não há suporte para as instruções *invoke* e *unwind*, que são utilizadas para implementar exceções, a função *switch*, que se trata de uma instrução de desvio para uma possível lista de destinos, e funções com argumentos variáveis (ZHAO et al., 2012).

3.2.2 Organização do *framework* VeLLVM

Uma parte desse trabalho foi analisar o funcionamento do *framework*, pois sua arquitetura não está documentada em nenhum lugar, dificultando a sua utilização para formalização de outras transformações. A única documentação disponível são arquivos *README.txt*, que explicam bastante brevemente o que alguns arquivos de código fonte devem conter, porém não dão uma visão geral de como o *framework* funciona. Por exemplo, a documentação apresentada não instrui como podemos usar o interpretador gerado pelo VeLLVM para rodar um código LLVM IR, ou onde podemos encontrá-lo.

Além da necessidade de entender o código fonte do VeLLVM, foi necessário avaliar se é possível utilizá-lo para implementar a transformação ACCE sem nenhuma ou pouca

modificação no *framework*.

Boa parte do código do VeLLVM é escrita utilizando Coq. O Coq é um provador de teoremas que provê uma linguagem para escrever definições matemáticas, algoritmos executáveis e teoremas. Utilizando essa linguagem, provas podem ser desenvolvidas e verificadas por um computador, e algumas provas podem ser geradas automaticamente pelo próprio provador (BERTOT; CASTÉRAN, 2004).

Entre o código escrito utilizando Coq, está a definição das semânticas formais, do sistema de tipos e da memória. Também foi desenvolvida em Coq uma definição que permite, dada uma semântica, executar um programa para essa semântica.

Uma funcionalidade interessante que o Coq oferece é a exportação do código para OCaml, Haskell ou Scheme. O VeLLVM utiliza essa funcionalidade, e complementa o código escrito em Coq com uma parte escrita em OCaml. Obviamente, todas as provas são construídas a partir do código Coq, sendo o código OCaml apenas para interface com o LLVM e o usuário.

Entre as funcionalidades escritas em Ocaml está um pequeno *wrapper* do código Coq para que seja possível executar llvm *bytecode*. A execução é feita através de uma ferramenta chamada simplesmente de *Interpreter*. O LLVM provê *bindings* em OCaml de suas funções de baixo nível, escritas em C, e o código do VeLLVM se aproveita disso em diversos pontos, como por exemplo para fazer *parsing* do LLVM *bytecode*. Também é escrito em OCaml o código necessário para transformar o código em LLVM *bytecode* nas estruturas definidas em Coq. Na figura 3.2 temos uma visão geral da arquitetura do VeLLVM, inclusive com exemplos de funções utilizadas diretamente dos bindings de OCaml do LLVM.

3.2.3 Definição de identificadores

O código escrito em Coq não diferencia entre os diferentes tipos de identificadores disponíveis no LLVM IR. Para ele, os identificadores de variáveis locais, que são precedidos por %, os de variáveis globais, que são precedidos por @, e as labels, que não têm um caractere predecessor, são todos iguais.

Todos os identificadores e variáveis são do tipo *atom*, definidos em uma das dependências do VeLLVM, a biblioteca *Metatheory*. *Atoms* são objetos sem estrutura que podem ser gerados sob demanda, sendo que sempre podemos definir a igualdade de dois *atoms* (AYDEMIR; CHARGUERAUD, 2012).

Um ponto importante para essa implementação referente aos *atoms* é que se não houver uma referência para ele (ou uma definição que gere sempre o mesmo *atom*), não é possível referenciar um *atom*, pois são todos encarados como objetos sem estrutura. Isso, na prática, significa que a partir do código Coq é impossível procurar por uma função pelo seu nome. Por exemplo, não é possível encontrar a função "*main*".

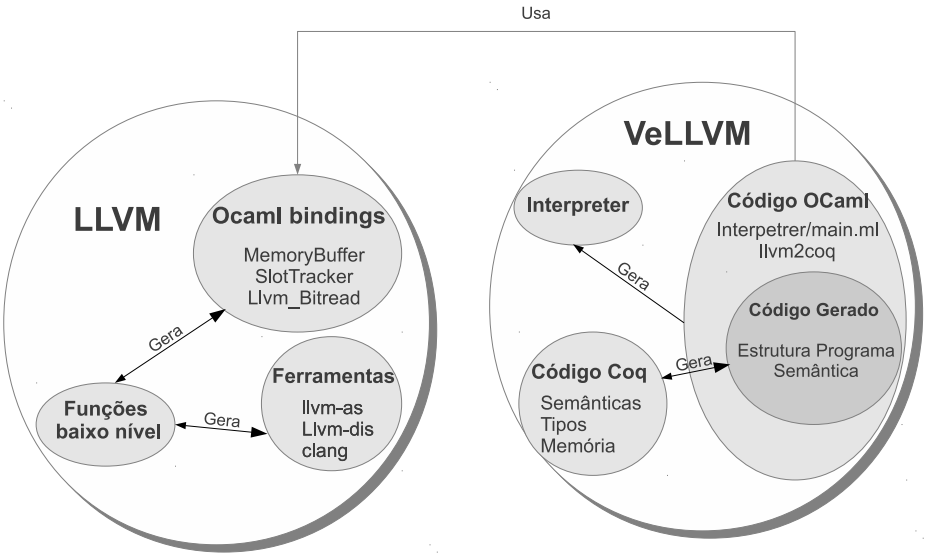

Figura 3.2: Organização do framework VeLLVM e relação com o código do LLVM

Como observado em um dos e-mails enviados por Zhao durante a construção desse trabalho, é possível descobrir qual é a função *main* caso o programa esteja sendo interpretado, pois ela está sempre na base da pilha utilizada pelo interpretador. Porém, ainda não é possível definir a função *main* em tempo de compilação. Note que uma prova adicional deve ser construída para se ter certeza que a função *main* sempre é a primeira da pilha.

3.2.4 Substituições de definições na tradução de Coq para Ocaml

No processo de tradução do código Coq para OCaml, algumas funções são substituídas por uma implementação alternativa em OCaml. Essas substituições são necessárias para poder interpretar o código em OCaml.

Como exemplo de uma substituição, a função que gera um novo *atom*, descrita em 3.2.3, é substituída por uma função que retorna a string $\%\_var\_X$, onde X é o número de vezes que a função foi chamada. Isso é necessário porque ao utilizar a interface OCaml do LLVM para carregar o *bytecode* e transformar em LLVM IR, todos os identificadores são strings, por exemplo "@main"ou então "%_var_22".

Essa re-definição dos *atoms* também faz com que seja possível localizar a função com nome "main"ou qualquer outro nome conhecido quando utilizando o código Coq já transformado para OCaml. Isso também é necessário para definir por onde o código será executado.

4 ACCE NO VELLVM

4.1 Preparação do ambiente

Para implementar a transformação ACCE utilizando o *framework* VeLLVM foi necessário preparar um ambiente de desenvolvimento. O código do framework VeLLVM está evoluindo muito rapidamente, para suportar versões mais recentes do LLVM, Coq e OCaml. O primeiro problema a ser resolvido com esse trabalho foi criar um ambiente para a última versão do VeLLVM. Para isso foi utilizada uma VM criada com VirtualBox, ferramenta para criação e gerenciamento de VMs. Como era conhecido que outros pesquisadores estavam passando por dificuldades de configuração do ambiente, contribuímos com o VeLLVM disponibilizando na web uma VM configurada para compilação e edição de código. Até o momento, ao menos o grupo de pesquisa de semântica da *University of Illinois at Urbana-Champaign* (UIUC) se beneficiou dessa ação. Segundo relatos por e-mail, estavam a ponto de desistir de utilizar o framework por ter gasto tempo demais tentando arrumar o ambiente de programação.

A documentação de instalação do VeLLVM é imprecisa e incompleta. Por exemplo, o *README.txt* cita arquivos que não existem, e não é informado ao usuário a necessidade de copiar arquivos manualmente em conjunto com a utilização da ferramenta *Make*. Foi necessário trocar 28 e-mails com o autor de (ZHAO et al., 2012) para conseguir completar a instalação do *framework*. Essa troca de mensagens foi útil tanto para o desenvolvimento desse trabalho quanto para completar a documentação de instalação do VeLLVM, pois no final dele foi possível notificar ao autor algumas modificações que devem ser feitas nas instruções de instalação. Há também uma séria limitação das versões de LLVM, Coq e Ocaml que devem ser utilizadas.

A VM disponibilizada para uso possui Ubuntu 12.04 com OCaml 3.12.1. O Coq utilizado foi da versão 8.3pl1, compilado no ambiente. Também foi instalado o LLVM versão 3.0 com uma série de modificações disponibilizadas em conjunto com o próprio LLVM, que são necessárias para poder utilizar o *framework*.

Para auxiliar o desenvolvimento, foi instalado o editor de textos vim, com *plugins* e configurações para melhor editar arquivos de código Coq e OCaml. Também foi instalado o editor de textos emacs e o seu *plugin* proofgeneral, que facilita na avaliação dos arquivos Coq.

4.1.1 Modificações no VeLLVM

A transformação ACCE requer adição de código. Essa adição não somente se limita à adição de comandos em um bloco. É necessário definir uma função global (GEH) e várias *labels* em tempo de compilação. O VeLLVM originalmente somente dá suporte à inserção de identificadores de variáveis locais.

Como mencionado em 3.2.4, os identificadores globais, locais e *labels* têm todos o mesmo tipo na implementação do VeLLVM: são todos do tipo *Atom*. Ao transformar o código Coq em OCaml, os dados do tipo *Atom* são transformados no tipo *String*, e todos os *atoms* são convertidos para *strings* iniciadas com o caractere %, logo, todas são tratadas como variáveis locais.

Não podemos apenas dividir o tipo *Atom* em outros três tipos abstratos diretamente no código Coq do VeLLVM, pois todas as provas já existentes no VeLLVM utilizam esses elementos como *Atoms*. O tipo *Atom* não é introduzido pelo VeLLVM. Esse conceito é utilizado pela biblioteca *Metatheory*.

Para contornar esse problema foi necessário modificar diretamente a biblioteca *Metatheory*. Nela, para gerar um *Atom*, utilizamos o *Parameter atom_fresh_for_list*, que recebe uma lista qualquer e gera um *Atom*. O conceito de *Parameter* de um tipo em Coq é semelhante ao de construtor em programação orientada a objetos. A lista é passada vazia para que possa ser gerada uma referência, pois no Coq, assim como em muitas linguagens funcionais, métodos sem nenhum parâmetro são simplificados.

O código inserido na biblioteca é a dupla replicação do *Parameter atom_fresh_for_list*, nomeados como *new_global_id* e *new_label*. Enquanto o *Parameter* original mantém a criação de variáveis locais, os dois *Parameters* inseridos visam gerar variáveis globais e labels. Note que como são *parameters* duplicados, todas as provas feitas para *atom_fresh_for_list* também são aplicáveis para *new_global_id* e *new_label* (e de fato, duplicadas também na modificação). Note que no ponto de vista semântico do Coq, podemos utilizar qualquer um dos *parameters* que não fará diferença alguma, pois todos tem tipo *Atom*, com o mesmo algoritmo de criação.

Ao transformar o programa Coq em OCaml, os novos *parameters* são substituídos por funções, escritas em OCaml, que geram uma string com um formato válido. Assim, ao utilizar os *parameters atom_fresh_from_list*, *new_global_id* e *new_label*, o Coq encara todos eles retornando uma instância do tipo *Atom*, porém ao transformar o código Coq em OCaml, o tipo *Atom* é visto como *String* e esses métodos geram strings diferentes utilizando as regras de formação de variáveis locais, variáveis globais e *labels*.

4.2 Implementação

O artigo original da técnica ACCE (VEMU; GURUMURTHY; ABRAHAM, 2007) apresenta o código que deve ser inserido e algumas restrições nos valores de assinaturas, porém não apresenta um algoritmo detalhado para chegar no resultado final. Este capítulo

explica os algoritmos utilizados para implementar a técnica ACCE no VeLLVM. Todos os algoritmos seguem um mesmo padrão: primeiro os dados extraídos do programa são organizados em uma estrutura descrita, a seguir; e, em seguida, os dados originais são modificados de acordo com a estrutura gerada.

Apesar das restrições previstas pela técnica ACCE estarem descritas rapidamente neste capítulo, existe uma descrição mais detalhada no capítulo 2.3.

4.2.1 Geração de valores únicos

Grande parte das restrições para os valores de assinatura do ACCE leva em consideração apenas que valores devem ser iguais ou diferentes. A única exceção é o cálculo da metade inferior do NS dos blocos do tipo A e do NES de seus predecessores, que exigem que a assinatura siga um certo parâmetro em nível de bits.

O artigo original também não define o que deve ser feito quando não existem mais valores únicos disponíveis para serem utilizados como assinatura. Nesse caso, por simplicidade, definimos que utilizaremos um valor repetido, diminuindo assim a eficiência do algoritmo para programas muito grandes, porém mantendo qualquer programa compilável. Outro ponto importante ao definir uma função que gera a próxima assinatura é que o Coq deve ser capaz de provar que toda recursão acaba, ou então o programa escrito em Coq não é válido.

Para calcular o próximo valor de assinatura único, dois parâmetros são informados: o último valor único gerado e uma lista de valores que não podem ser escolhidos. O valor de assinatura gerado é o sucessor do ultimo valor, exceto se esse valor estiver na lista de valores proibidos, caso em que a função é chamada recursivamente. Como é necessário sempre ter uma garantia que a recursividade irá acabar, é passado também o número máximo de tentativas da recursão, que no caso é o numero de valores disponíveis para a assinatura. O código Coq dessa parte da implementação pode ser visto no bloco 4.1.

```
1  Fixpoint get_next_sig (sig:Z) (blacklist:list inf_bits) (max_try:nat) :
       Z :=
2  match max_try with
3  | S n =>
4    let next_sig := sig + 1 in
5    if sig_is_in_blacklist next_sig blacklist then
6      get_next_sig (next_sig + 1) blacklist n
7    else
8      next_sig
9  | _ => sig
10 end.
```

Bloco de código 4.1: Função que gera valores únicos

4.2.2 Cálculo das assinaturas

Conforme descrito em 2.3, para cada bloco devem ser calculados dois valores de assinaturas: o valor de NS, que é a assinatura de entrada, e o valor de NES, que é a

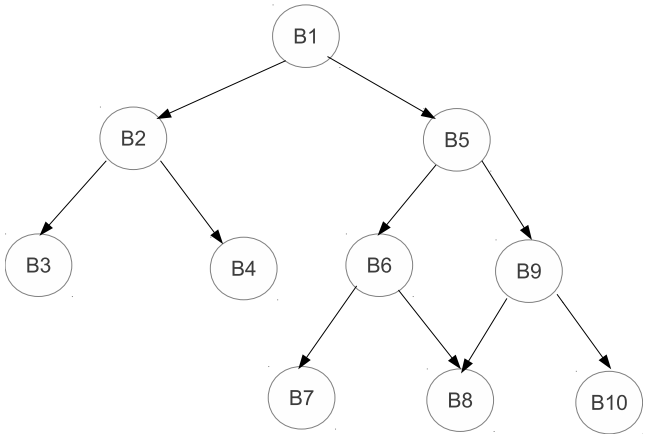

Figura 4.1: Exemplo de grafo de blocos básicos

assinatura de saída. Cada uma dessas assinaturas pode ser dividida em duas partes, superior (composta da metade dos bits mais significativos) e inferior (composta da metade dos bits menos significativos). Como descrito no artigo original, as restrições para os valores das assinaturas levam em consideração o tipo do blocos, que pode ser do tipo X ou do tipo A.

4.2.2.1 Cálculo do tipo de um bloco

Cada bloco deve ser definido como do tipo X ou A. Para cada bloco visitado, procura-se os seus sucessores e para cada um é testada a existência de mais de um predecessor. Caso exista, é um bloco do tipo A, caso contrário, do tipo X. O bloco de código 4.2 mostra a implementação utilizada.

```
1 Definition define_node_type (llvm_block:block) (llvm_blocks:list block)
      : node_type :=
2 if gt_dec (List.length (predecessors llvm_block llvm_blocks)) 1 then
3   if gt_dec (List.length (filter_by_multiple_successors (predecessors
        llvm_block llvm_blocks))) 0 then
4     type_A
5   else
6     type_X
7 else
8   type_X.
```

Bloco de código 4.2: Função que define o tipo de um bloco

4.2.2.2 Cálculo da parte superior

O cálculo da parte superior da assinatura é independente do tipo do bloco. O valor para a parte superior do NS deve ser único, e a parte superior do NES de seus predecessores deve ter o mesmo valor único. Porém, se um bloco $B1$ tem como sucessores os blocos $B2$ e $B3$, devemos ter $NES_u(B1) = NS_u(B2) = NS_u(B3)$ para conseguir manter a restrição.

Para conseguir manter essa restrição, é criada uma lista de duplas net, em que o primeiro valor é um conjunto de blocos que tem um predecessor em comum, e o segundo valor é a união dos conjuntos de blocos predecessores de cada bloco do primeiro valor. Por exemplo, para figura 4.1 temos a seguinte *nets*:

$$[net(\{B1\},\{\}), net(\{B2,B5\},\{B1\}), net(\{B3,B4\},\{B2\}),$$

$$net(\{B6,B9\},\{B5\}), net(\{B7,B8,B10\},\{B6,B9\})].$$

Para gerar essa lista de *nets*, os blocos do LLVM são visitados um por um. Caso nenhum de seus sucessores seja sucessor de uma *net*, uma nova entrada é criada na lista. Caso contrário, esse nodo e seus sucessores são adicionados nessa entrada da net. O algoritmo utilizado no programa Coq pode ser visualizado no bloco de código 4.3

```
1  Fixpoint create_nets (blocks: list acce_block) (nets : list net) (
       all_blocks: list acce_block) : list net :=
2  match blocks with
3  | head::tail =>
4    if (eq_dec (List.length (block_successors head all_blocks)) 0) then
5      if (has_one_on_net (block_successors head all_blocks) nets) then
6        create_nets tail (merge_on_net head nil nets all_blocks)
             all_blocks
7      else
8        let new_net_entry := net_intro (block_successors head all_blocks)
             (head::nil) in
9        create_nets tail (new_net_entry::nets) all_blocks
10     else
11      create_nets tail nets all_blocks
12 | nil => nets
13 end.
```

Bloco de código 4.3: Função que monta a lista de nets utilizada para definir as assinaturas superiores

Com as *nets* geradas, o trabalho de criar as assinaturas fica bem simplificado. Para cada *net*, os valores de NS_u dos blocos da primeira posição são iguais entre si, e iguais ao valor de $NESu$ dos blocos da segunda posição, diferentes das outras nets. Então, nesse ponto, é gerada uma assinatura única para cada net encontrada, conforme o bloco de código 4.4.

```
1  Fixpoint calc_upper_sigs (nets:list net) (sig:Z): prod Z (list net) :=
2  match nets with
3  | head::tail =>
4    let next_sig := get_next_sig sig nil max_try in
5    let (first_pos, sec_pos) := head in
6    let first_nets := set_ns_upper_sig first_pos next_sig in
7    let sec_nets := set_nes_upper_sig sec_pos next_sig in
8    let (last_sig, tail_nets) := calc_upper_sigs tail next_sig in
9    pair last_sig ((nets_intro first_nets sec_nets) :: tail_nets)
10 | nil => pair sig nil
11 end.
```

Bloco de código 4.4: Dada as nets do programa, insere as assinaturas superiores

4.2.2.3 Cálculo da parte inferior

O cálculo da parte inferior da assinatura depende do tipo do bloco. Para os blocos do tipo X, temos que o valor do NS_l do bloco deve ser igual ao valor do NES_l de seus sucessores. Para blocos do tipo A a parte inferior deve ser diferente, de modo que o número de bits em 0 de NS_l seja igual ao número de predecessores do bloco. Já o NES_l dos predecessores desse bloco são diferentes entre si, com o valor base do NS_l, mudando apenas um bit 0 para 1.

A implementação desse cálculo para os blocos de tipo X é bastante semelhante a suas assinaturas superiores, vista na seção passada, por isso não será apresentada novamente.

Já para a implementação dos cálculos para os blocos do tipo A , é necessário gerar uma assinatura com essa restrição nos bits, sendo impossível sempre pegar o sucessor da última assinatura gerada. Esse valor é gerado para cada bloco do tipo A encontrado. Exatamente para esse caso que na geração de assinaturas, descrito em 4.2.2, é necessário passar uma lista de valores que não podem ser gerados sequencialmente, pois esses valores gerados não respeitam nenhuma ordem.

O valor da assinatura do NS_l para um bloco de tipo A é o próximo valor lexicográfico disponível para seu número de bits. Por disponível, entende-se que esse valor não está na lista de valores gerados e é maior que a última assinatura sequencial gerada. Por exemplo, na figura 4.1, o bloco $B8$ é do tipo A. Digamos que a parte inferior da assinatura tenha 6 bits. Como $B8$ tem dois pais, o valor gerado deve ter 4 bits em 1 e dois em 0. Na primeira tentativa, o bit gerado tem valor 111100. Caso esse valor já esteja utilizado, o próximo valor é gerado,que consiste em 111010. Caso esse também esteja utilizado, o próximo valor gerado é 111001, e assim por diante. Após isso, uma lista de assinaturas com as variações desses bits são geradas para os blocos predecessores.

Devemos lembrar que esse algoritmo é recursivo e não tem uma condição de parada reconhecida pelo Coq. De fato, ele pode não encontrar nenhum valor disponível. Para a implementação desse algoritmo, semelhantemente ao que foi feito com a geração da próxima assinatura válida, também foi definido um número máximo de tentativas para ser válido perante o Coq, como pode ser visto no bloco de código. Caso o número máximo de tentativas, que é definido como o maior número possível da assinatura, se esgote,

o último valor possível é retornado. Novamente, no artigo original não é definido um comportamento para caso todas as partes inferiores tenham sido tomadas. Nesta implementação, o que acontece é uma repetição da utilização das assinaturas, o que pode gerar o não-reconhecimento de um CFE. Uma alternativa para esse reconhecimento seria gerar assinaturas de tamanho variável, de acordo com o número de blocos do programa, mas por simplicidade essa implementação não prevê esse tipo de controle. Nesse caso, seria mais simples construir uma prova para um programa com um número limitado de blocos, e generalizar depois. Táticas para provas com essa implementação serão discutidas na sessão de trabalhos futuros.

```
1  Fixpoint step_inf_binary (last_try:inf_bits) (blacklist:list inf_bits)
       (max_try:nat) :=
2  match max_try with
3  | 0 =>
4    last_try
5  | S i =>
6    let next_try := next_inf_binary last_try in
7    if negb (is_in_blacklist next_try blacklist) then
8      next_try
9    else
10     step_inf_binary (next_inf_binary next_try) blacklist i
11 end.
```

Bloco de código 4.5: Gera o próximo valor de assinatura inferior com restrições de bits

Após isso, a assinatura gerada é atribuída para o valor de NS_l do bloco de tipo A, e as assinaturas geradas para os predecessores são atribuídas ao valor de NES_l para os predecessores desse bloco, como pode ser visto no bloco de código 4.6.

```
1  Fixpoint set_a_rule_inf (pairs_list:list inf_pair) (sig:Z) (all_blocks:
       list acce_block) (blacklist:list inf_bits) : list acce_block :=
2  match pairs_list with
3  | head::tail =>
4      let (blocks,parents) := head in
5      let (head_sig, parent_sigs) := take_inf_binary (Z_of_nat (
           List.length parents)) blacklist in
6      let new_blocks := apply_each_inf_ns blocks sig all_blocks in
7      let new_blocks2 := apply_each_inf_nes parents sig all_blocks in
8      let new_blacklist := head_sig :: blacklist in
9      set_a_rule_inf tail sig new_blocks2 new_blacklist
10 | nil => all_blocks
11 end.
```

Bloco de código 4.6: Atribui as assinaturas para blocos do tipo A

4.2.3 Adição de código

Para entender essa seção, é imprescindível o completo entendimento da seção 2.3, pois ela descreve o algoritmo que estamos implementado aqui. Diferentemente do cálculo das assinaturas, essa seção é bem mais direta em relação ao que é descrito no artigo original do ACCE, sem a necessidade de muitos algoritmos originais. As principais mudanças são em relação à nova linguagem-alvo, LLVM IR, que possui um conceito de blocos básicos

que não é presente na linguagem implementada no artigo original.

4.2.3.1 Adição de variáveis globais

Na etapa de adição de código, primeiramente são adicionadas as variáveis globais ao programa. Essas variáveis globais são referenciadas no código como var_S_atual, que guarda o valor da assinatura atual do sistema, que será testada no início do próximo bloco, var_F, que corresponde ao identificador da função atual, var_error_flag, que é uma variável global que indica se um erro está sendo tratado, e algumas outras auxiliares. O bloco de código principal para a inserção dessas variáveis é apresentado em 4.7.

```
1  Definition add_global_vars (mod5:module) (var_S_atual:id) (var_F:id) (
       var_error_flag:id) (sgeh:id) (num_error_feh:id) (var_num_error:id)
       : module :=
2    let (labs,defs,prods) := mod5 in
3    let var_S_atual_prod := prod_gvar_32 var_S_atual in
4    let var_F_prod := prod_gvar_32 var_F in
5    let var_error_flag_prod := prod_gvar_1 var_error_flag in
6    let sgeh_prod := prod_gvar_32 sgeh in
7    let num_error_feh_prod := prod_gvar_32 num_error_feh in
8    let var_num_error_prod := prod_gvar_32 var_num_error in
9    let new_prods := prods ++ var_S_atual_prod :: var_F_prod ::
         var_error_flag_prod :: sgeh_prod :: num_error_feh_prod ::
         var_num_error_prod :: nil in
10   module_intro labs defs new_prods
```

Bloco de código 4.7: Define variáveis globais

4.2.3.2 Modificação nos blocos básicos

Para essa etapa da transformação, é necessário notar a diferença de arquitetura da implementação do artigo que descreve o ACCE para a plataforma dessa arquitetura. Para a linguagem implementada no trabalho original, *labels* podem ser introduzidas em qualquer ponto do código. Já em LLVM IR, utiliza-se o conceito de bloco básico, que torna mais restritivo o lugar onde *labels* podem ser implementadas. Lembrando de 3.1.1 que o código deve ser construído em blocos básicos, e que um bloco básico é uma sequência de *label*, comandos e instrução terminal, usualmente uma instrução de *branch* ou *return*.

Levando essa restrição em consideração, cada bloco básico do programa original será transformado em 5 blocos básicos, exceto o primeiro bloco básico de cada função, que é transformado apenas em dois. Esses blocos básicos são referenciados no código como $checkpoint$, $data$, $data2$, $original$ e $update$. O primeiro bloco básico é transformado apenas em $original$ e $update$. A função responsável por dividir cada bloco em 5 está listada em 4.8. Além disso, para cada bloco básico também são adicionados blocos correspondentes à *FEH*, explicada na seção 4.2.3.3, e duas variáveis globais, $tmpcmp$ uma variável temporária para a modificação das instruções de $branch$, e $exec$, que define se o bloco já havia começado a executar e é utilizada na recuperação de erros.

Os blocos adicionados antes do código original são $checkpoint$, $data$ e $data2$. Esses blocos consistem no cabeçalho descrito na seção 2.3. O bloco $original$ é o bloco que con-

tém o código do bloco original com pequenas modificações previstas pelo ACCE, como, por exemplo, salvar o ID da função atual na variável global var_F ao retornar de uma chamada de função. Após o bloco original, há um bloco extra, $update$, que corresponde ao rodapé descrito na seção 2.3.

```
1  Fixpoint split_each_block (some_blocks:blocks) (signed_blocks:list
       acce_block) (feh_label:l) (var_S:id) (var_S_atual:id) (exec:id) :
       ((list block)*(list id))*(list (l*Z)) :=
2  match some_blocks with
3  | head :: tail =>
4    match head with pair head_label (stmts_intro _ _ term5) =>
5    match find_block_by_l head_label signed_blocks with
6    | Some signed_block =>
7      let this_ns := ns signed_block in
8      let this_nes := nes signed_block in
9      let exec := (projT1 (get_next_global nil)) in
10     let tmpcmp := (projT1 (get_next_global nil)) in
11     let checkpoint_label := head_label in (*(projT1 (get_next_label nil
           )) in *)
12     let original_label := (projT1 (get_next_label nil)) in
13     let data_label := (projT1 (get_next_label nil)) in
14     let data2_label := (projT1 (get_next_label nil)) in
15     let update_label := (projT1 (get_next_label nil)) in
16     let (tail_blocks_and_vars,tail_feh_data) := split_each_block tail
         signed_blocks feh_label var_S var_S_atual exec in
17     let (tail_blocks, tail_vars) := tail_blocks_and_vars in
18       pair (
19         pair (
20           create_checkpoint head signed_block checkpoint_label
                original_label exec feh_label var_S ::
21           create_data head signed_block original_label data2_label
                data_label exec var_S_atual ::
22           create_data2 head signed_block original_label data2_label
                var_S ::
23           create_original head signed_block update_label original_label
                exec feh_label var_S tmpcmp ::
24           create_update head signed_block update_label var_S term5
                tmpcmp ::
25           tail_blocks
26         )
27         (exec :: tmpcmp :: tail_vars)
28       )
29       ((pair data_label this_ns)::(pair update_label this_nes)::
           tail_feh_data)
30   | None => pair (pair nil nil) nil
31   end
32   end
33 | nil => pair (pair nil nil) nil
34 end.
```

Bloco de código 4.8: Divide um bloco em 5

O bloco $checkpoint$ é o primeiro bloco a executar no programa, e sua função é testar a assinatura. Se a assinatura não corresponde ao esperado, o fluxo do programa é desviado

para a função de tratamento de erro local, FEH. Além disso, esse bloco seta a variável $exec$ para $false$, indicando que o bloco ainda não começou a ser executado.

O bloco $data$ é responsável por guardar a nova assinatura atual. No caso de uma execução normal, o programa é desviado diretamente para o bloco original, porém em caso de erro ele é desviado para o $data2$. Essa decisão é baseada na variável $exec$, comentada anteriormente. O bloco básico $data2$ apenas atualiza uma variável global que informa na recuperação de um erro que o erro foi recuperado atualizando a assinatura atual e continua a execução.

O bloco $original$ começa setando $exec$ para $true$, indicando que o bloco começou a ser executado. Além disso, grava a assinatura atual em uma variável local. Em seguida executa o código original. Para finalizar, verifica se a assinatura mudou - em caso afirmativo, redireciona o fluxo para a função FEH e, em caso negativo, continua o fluxo esperado do programa, desviando para $update$.

O bloco $update$, por fim, atualiza a assinatura, que deve ser esperada no início do próximo bloco, no bloco $checkpoint$, e inclui o desvio do bloco básico original, só que ao invés de redirecionar para o bloco previsto no programa original, o desvio ocorre para o bloco $checkpoint$ correspondente do bloco original.

4.2.3.3 Adição da função FEH

Como mencionado na seção anterior, além das transformações nos blocos básicos do programa, também é necessário adicionar um conjunto de blocos básicos FEH para cada bloco, responsável por tratar erros localmente e em caso de falha delegar a tarefa para a função GEH.

O número de blocos básicos inseridos para definir a FEH é variável, de acordo com o número de blocos básicos da função em que é inserida. Isso se dá pois há um teste seguido de desvio para cada bloco básico da função, utilizado durante a recuperação de um erro local. Todos esses blocos extras poderiam ser transformados em apenas 1 utilizando a função *switch* disponível na LLVM IR, porém o VeLLVM não implementa a semântica dessa instrução. No total são adicionados $7 + n$ blocos básicos para a FEH, onde n é o número de blocos básicos da função no código original. O bloco de código que adiciona esses blocos básicos pode ser visto em 4.9.

O primeiro bloco adicionado é o $entry$, que é responsável por verificar se há um erro em andamento. Caso positivo, redireciona para ats, caso negativo para $values$. O bloco ats é responsável por recuperar o valor da assinatura em caso de erro global, e em seguida redireciona para $values$.

O bloco $values$ verifica se o erro ocorreu enquanto dentro da função atual ou se o erro ocorreu enquanto executando outra função. Caso o erro tenha ocorrido enquanto executando outra função, a função GEH é chamada (através do bloco $call\_geh$), caso contrário o fluxo é redirecionado para $check\_n\_error$.

```
1  Definition add_feh (feh_data:list (l*Z)) (feh_entry:id) (all_blocks:
       list block) (var_S:id) (var_S_atual:id) (var_F:id) (var_FID:id) (
       var_error_flag:id) (geh:id) (sgeh:id) (num_error_feh:id) : list
       block :=
2  let feh_ats := (projT1 (get_next_label nil)) in
3  let feh_values := (projT1 (get_next_label nil)) in
4  let feh_check_n_error := (projT1 (get_next_label nil)) in
5  let feh_call_geh := (projT1 (get_next_label nil)) in
6  let feh_exit := (projT1 (get_next_label nil)) in
7  let feh_search_n := (projT1 (get_next_label nil)) in
8  let feh_switch := (projT1 (get_next_label nil)) in
9  let feh_switch_cond := (projT1 (atom_fresh_for_list nil)) in
10
11 all_blocks ++
12 add_feh_entry feh_entry feh_ats feh_values var_error_flag ::
13 add_feh_ats feh_ats feh_values sgeh var_S::
14 add_feh_values feh_values feh_check_n_error feh_call_geh var_S
       var_S_atual var_F var_FID sgeh::
15 add_feh_call_geh feh_call_geh geh ::
16 add_feh_check_n_error feh_check_n_error feh_exit feh_search_n
       var_error_flag num_error_feh::
17 add_feh_search_n feh_search_n feh_switch feh_switch_cond var_S ::
18 add_feh_exit feh_exit ::
19 add_feh_switch feh_data feh_switch feh_switch_cond feh_entry var_S.
```

Bloco de código 4.9: Adição dos blocos básicos que representam a FEH

O bloco $check\_n\_error$ é a entrada para tentar descobrir em que bloco o erro foi começado. Em caso de o erro não ser encontrado posteriormente, essa função é chamada novamente. Em caso de ela ser chamada 3 vezes consecutivas sem resolver o erro, o programa é terminado graciosamente. O número 3 foi sugerido pelo autor do artigo original sobre a técnica. Em caso do programa acabar, o bloco feh_exit é chamado, que simplesmente acaba o programa. Em caso contrário, redireciona para o primeiro bloco do grupo de blocos $switch$.

O conjunto de blocos $switch$ é responsável por encontrar o bloco em que o erro ocorreu, testando sucessivamente a assinatura atual com as assinaturas NS dos blocos básicos da função.

4.2.3.4 Adição da função GEH

A GEH é uma função chamada por um dos conjuntos de blocos nominados FEH, presentes para cada função do programa, caso conclua que o erro foi gerado em outra função do programa.

Semelhantemente à FEH, a GEH também tem um número de blocos básicos variáveis, de acordo com o número de funções do programa. O número de blocos básicos dessa função é $3 + n$, onde n é o número de funções do programa. O código responsável por gerar os blocos básicos dessa função pode ser visto em 4.10.

```
1  Definition geh_blocks (var_F:id) (var_error_flag:id) (num_error:id) (
       fdefs:list fdef) : blocks :=
2  let geh_entry_lab := projT1 (get_next_label nil) in
3  let find_function_lab := projT1 (get_next_label nil) in
4  let check_num_error_lab := projT1 (get_next_label nil) in
5  let exit_lab := projT1 (get_next_label nil) in
6  let temp_F := projT1 (atom_fresh_for_list nil) in
7  let switch_data := generate_switch_data 1 fdefs in
8  let geh_entry := create_geh_entry geh_entry_lab find_function_lab var_F
        var_error_flag in
9  let find_function := create_find_function_switch switch_data
       find_function_lab check_num_error_lab temp_F in
10 let check_num_error := create_check_num_error check_num_error_lab
       exit_lab find_function_lab var_F num_error var_error_flag in
11 let exit := create_exit exit_lab num_error in
12 geh_entry :: nil
13 ++ find_function ++
14 check_num_error :: exit :: nil.
```

Bloco de código 4.10: Criação dos blocos básicos da função GEH

O primeiro bloco a ser chamado é o $entry$, que atribui $true$ a variável global $error\_flag$ indicando que uma recuperação de erro global está sendo iniciada, logo em seguida redireciona para o conjunto de blocos $check\_num\_error$. Esse conjunto de blocos procura a função atual do sistema comparando a variável global var_F com a constante definida para cada função. Caso o valor de var_F não corresponda à constante de nenhuma função, finaliza o programa. Caso encontre a função correspondente, redireciona o programa para o bloco feh_entry da função, para que o erro seja tratado localmente.

5 CONCLUSÃO

5.1 Resultados Obtidos

Este trabalho resultou em uma base concreta para que provas a respeito da transformação ACCE possam ser construídas em Coq, utilizando o *framework* VeLLVM. Também provê algoritmos para alcançar as restrições descritas no artigo original da técnica, que podem ser aplicados em qualquer linguagem de programação.

O *framework* VeLLVM também foi beneficiado com este trabalho, pois foi criado um método para que transformações que adicionam identificadores globais e *labels* também possam ser implementadas nesse *framework*. Além disso, através da experiência de instalação, foi possível melhorar a documentação existente do *framework*, além de escrever um breve relato de seu funcionamento e organização. Podemos citar também como um ponto positivo a criação de uma VM disponibilizada na internet para que qualquer pessoa possa utilizar o *framework* sem se preocupar com a criação de um ambiente, que se provou bastante trabalhoso durante o desenvolvimento destSe trabalho.

5.2 Trabalhos futuros

Uma etapa que evidentemente ficou em aberto com a conclusão deste trabalho é a construção de provas a respeito da técnica ACCE. Uma primeira prova a ser construída, a mais simples, consiste no caso em que nenhuma falha ocorre. Nesse caso, um programa submetido à transformação deve ser equivalente a um programa sem nenhuma transformação.

Para construir uma prova de que dois programas são equivalentes, o primeiro passo é definir uma noção de equivalência. Caso a definição de equivalência seja a formalização da idéia "Dois programas são equivalentes se e somente se o resultado externo for o mesmo para os dois programas", uma possível tática para construir a prova consiste em eliminar o código inserido pela transformação. Levando em consideração que o sistema não apresente erros, deve ser possível provar que para todo programa escrito em LLVM IR, ele nunca é desviado para os blocos inseridos na etapa de FEH, e então esses podem ser removidos e as instruções de desvio simplificadas. Retirando os blocos que constituem o FEH, não haverá mais nenhuma chamada para GEH, então essa função também pode ser removida. Por fim, deve-se mostrar que as instruções restantes incluídas nos

blocos básicos podem ser resumidas em instruções de *load*, *store* e operações sobre as variáveis incluídas na própria transformação, não interferindo no resultado externo esperado do programa, podendo ser removidas sem nenhum prejuízo.

Uma prova mais complexa diz respeito à eficácia da técnica quanto à sua capacidade de detecção e correção de falhas. Essa prova depende em primeiro lugar da formalização de um modelo de falha. Esse modelo deve refletir o tipo de falha que ACCE se propõe a detectar e corrigir, por exemplo, que sempre haverá um erro de controle de fluxo logo após a primeira instrução de *jump* em um programa. A partir daí, deve-se enumerar as possibilidades de locais onde o desvio pode chegar, e provar que o programa consegue se recuperar para determinados casos. Outra prova com o mesmo modelo de falhas pode ser construída mostrando que para outros determinados casos o programa não mantém a mesma semântica.

Como um exemplo intuitivo vamos definir como modelo de falha uma falha causada antes da primeira instrução do primeiro bloco de $checkpoint$ de um programa. Caso essa falha faça com que a próxima instrução seja exatamente a primeira do primeiro bloco da função GEH, o programa intuitivamente deve conseguir se recuperar. Agora, se ele for desviado para logo antes de soma em uma variável já executada no primeiro bloco do programa, o programa transformado possivelmente não conseguirá manter a semântica do programa original. Como um exemplo de programa que possa ter a semântica alterada nesse caso, podemos pensar em um programa simples que soma dois números e mostra na tela o resultado e se ele é um número par ou ímpar. O programa teria 3 blocos básicos, o primeiro onde os números são somados, seguido de um desvio condicional para os dois outros blocos: Um imprime o número e "par", e outro que imprime o número e "ímpar". Nesse caso, se o programa fosse desviado para antes da instrução de soma, esses números seriam somados duas vezes e, apesar de após a recuperação do programa ele conseguir imprimir "par"ou "ímpar"corretamente na tela, o número a ser impresso seria muito maior.

REFERÊNCIAS

AVIZIENIS, A. et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Trans. Dependable Secur. Comput.**, Los Alamitos, CA, USA, v.1, n.1, p.11–33, Jan. 2004.

AYDEMIR, B.; CHARGUERAUD, A. **Documentação de Atoms da biblhoteca Metatheory**. Disponível em: <http://www.cis.upenn.edu/ plclub/popl08-tutorial/code/coqdoc/Atom.html>. Acesso em: dec 2012.

BERTOT, Y.; CASTÉRAN, P. **Interactive Theorem Proving and Program Development. Coq'Art**: the calculus of inductive constructions. [S.l.]: Springer Verlag, 2004. (Texts in Theoretical Computer Science).

BHATTACHARYA, K.; RANGANATHAN, N. A unified gate sizing formulation for optimizing soft error rate, cross-talk noise and power under process variations. In: QUALITY OF ELECTRONIC DESIGN, 2009. ISQED 2009. QUALITY ELECTRONIC DESIGN. **Anais...** [S.l.: s.n.], 2009. p.388 –393.

HUA, B.; XU, B.; GAO, Y. Explicitly typed static single-assignment form. In: EDUCATION TECHNOLOGY AND COMPUTER (ICETC), 2010 2ND INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. v.1, p.V1–43 –V1–47.

IROCTECH. **Página da empresa IROCTech**. Disponível em: <http://iroctech.com/>. Acesso em: dec 2012.

LEROY, X. Formal verification of a realistic compiler. **Communications of the ACM**, [S.l.], v.52, n.7, p.107–115, 2009.

MUKHERJEE, S. **Architecture Design for Soft Errors**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

NAGARAKATTE, S. et al. SoftBound: highly compatible and complete spatial safety for c. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2009. **Proceedings...** [S.l.: s.n.], 2009.

PARIZI, R. B. Impact on Reliability in the Control-Flow of Programs under Compiler Optimizations. **Simpósio Brasileiro de Engenharia de Sistemas Computacionais**, [S.l.], 2012.

TANG, H. H. K. Nuclear physics of cosmic ray interaction with semiconductor materials: particle-induced soft errors from a physicist's perspective. **IBM Journal of Research and Development**, [S.l.], v.40, n.1, p.91 –108, jan. 1996.

TEAM, L. **Página inicial do grupo LLVM**. Disponível em: <http://llvm.org/>. Acesso em: dec 2012.

VEMU, R.; GURUMURTHY, S.; ABRAHAM, J. ACCE: automatic correction of control-flow errors. In: TEST CONFERENCE, 2007. ITC 2007. IEEE INTERNATIONAL. **Anais...** [S.l.: s.n.], 2007. p.1 –10.

WEN, S. et al. Thermal neutron soft error rate for SRAMS in the 90NM - 45NM technology range. In: RELIABILITY PHYSICS SYMPOSIUM (IRPS), 2010 IEEE INTERNATIONAL. **Anais...** [S.l.: s.n.], 2010. p.1036 –1039.

ZHAO, J. et al. Formalizing the LLVM intermediate representation for verified program transformations. **SIGPLAN Not.**, New York, NY, USA, v.47, n.1, p.427–440, Jan. 2012.