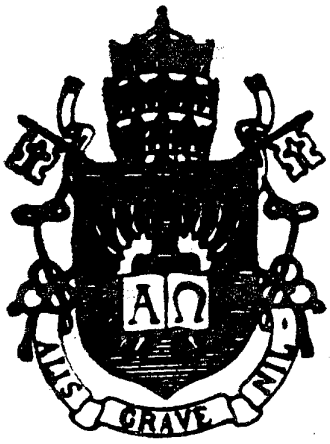

PUC

Série: Monografias em Ciência da Computação,
No. 10/90

TRATAMENTO DE EXCEÇÕES

Noemi R. Rodriguez

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC Rio - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, 10/90
Editor: Paulo A. S. Veloso

Agosto, 1990

TRATAMENTO DE EXCEÇÕES*

Noemi R. Rodriguez

* Apresentado pelo Prof. José Lucas M. Rangel Netto

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

Resumo

O termo "exceção" é usado por vários autores para descrever determinadas situações que ocorrem durante a execução de um programa. Este trabalho expõe os diversos conceitos do termo existentes na literatura e as principais soluções para seu tratamento, discutindo suas vantagens e desvantagens. Também são discutidos a necessidade e papel de mecanismos lingüísticos especialmente voltados para o tratamento de exceções.

Palavras Chave:
tratamento de exceções, linguagens de programação, erro de execução, falha

Abstract

The word "exception" is used by many authors to describe certain situations which arise during the execution of a program. In this report we explore the different visions of the term which can be found in the literature and the main solutions proposed for the handling of these situations. The need and role of linguistic mechanisms specially defined for the handling of exceptions is also discussed.

Keywords:
exception handling, programming languages, execution error, failure

Índice

I. Introdução

Ao programar uma rotina, sempre nos deparamos com situações pouco freqüentes ou onde não há o que fazer. Alguns exemplos típicos são: divisão por zero, overflow, leitura de EOF em um arquivo e estouro de memória. O problema é que gostaríamos de supor que os dados satisfazem determinadas assertivas que, em geral, não podem ser verificadas estaticamente. A violação de tais propriedades, cujo tratamento pode ser a parte mais tediosa, ainda que menos valorizada, do desenvolvimento de um programa, é o que intuitivamente chamamos de exceção. Coloca-se então a pergunta: que facilidades uma linguagem pode oferecer de forma a facilitar o trabalho do programador no momento de tratar este tipo de violação?

Começando com a definição de PL/1, vários trabalhos têm se preocupado com a criação de mecanismos linguísticos para o tratamento de casos excepcionais. O objetivo deste trabalho é tentar estabelecer que tipos de casos excepcionais realmente merecem um mecanismo especial e quais as vantagens e desvantagens dos principais mecanismos propostos.

Quando falamos em mecanismos linguísticos para tratamento de exceções, estamos nos referindo a mecanismos que seguem a idéia lançada por PL/1. Determinados estados de programas, considerados "anormais" segundo algum critério, são associados com identificadores que chamaremos de condições. Nos pontos do programa onde se verifica que um destes estados excepcionais foi atingido são inseridas sinalizações do identificador associado, e o programador tem maneiras de definir um conjunto de comandos a ser executado no caso de ocorrência de uma condição. Este conjunto de comandos é denominado um tratador de exceção. A amplitude e indefinição desta descrição é proposital, uma vez que existem várias visões distintas, que serão discutidas na seção II, de como deve ser o comportamento de um tal mecanismo. Outras técnicas relacionadas a recuperação de erros e tolerância a falhas, como os blocos de recuperação propostos em [Randell 75], não serão consideradas neste trabalho.

A discussão sobre o que é exatamente um estado excepcional nos parece especialmente interessante. Parece indiscutível que, por exemplo, uma divisão por zero é uma tal condição: não se espera que um programa teste se uma variável é diferente de zero antes de cada operaçao de divisão e nem se admite uma implementação de divisão que permita que a máquina "voe" no caso do divisor ser zero. é necessário então algum mecanismo que permita um "fim digno" para o programa em um caso como este. Por outro lado, uma rotina de busca caracterizar a ausência do valor procurado como exceção já é algo bem mais complicado. Dependendo do contexto em que esta rotina é usada, a ausência do símbolo pode ou não ser "anormal".

Analisaremos primeiramente, na seção II, como os trabalhos existentes colocam o conceito de exceção, e apresentaremos uma classificação de possíveis condições de exceção. Na seção III, veremos de que forma estes casos poderiam ser tratados sem a inclusão de um mecanismo especial . A seção IV apresenta os mecanismos de exceção estudados, comparando-os em relação a um conjunto de pontos considerados relevantes. A partir da discussão desses pontos e da consideração das alternativas apresentadas na seção III, a seção V discute o de mecanismo de tratamento de exceção considerado adequado.

II. Os diversos conceitos de exceção

II.1. As primeiras propostas

PL/1 foi a primeira linguagem de programação a propor o conceito de tratamento de exceções. Noble [Noble 68] afirma que as condições de PL/1 normalmente indicam que o programa excedeu alguma limitação da máquina. No entanto, como observado por Black [Black 82], apenas algumas poucas das 18 condições definidas em PL/1 referem-se realmente a casos deste tipo. Noble também diz: "Na prática muitas das condições serão usadas simplesmente como um meio de detectar erros como índice inválido, divisão por zero, etc, mas algumas das condições são projetadas para serem usadas como métodos normais para controlar o fluxo de programação em

circunstâncias do programa que são normalmente esperadas em algum momento mas imprevisíveis, por exemplo, ENDFILE, ENDPAGE." Resumindo, pode-se inferir que o mecanismo de condições foi pensado para controlar situações (a) onde se excedeu alguma limitação da máquina, (b) onde uma operação foi invocada com entradas inadequadas (indexação de array ou divisão, nos exemplos dados) e (c) normais (onde condições são usadas como um método para facilitar a programação).

Um dos primeiros trabalhos em que se encontra uma discussão do conceito de exceção é o de Goodenough [Goodenough 75]. Goodenough diz que condições de exceção são aquelas das quais uma operação avisa seu invocador. Como já dito em [Black 82], esta definição abrange qualquer resultado ou efeito observável da operação. Além desta definição, Goodenough dá um conjunto de usos para exceções que nos fornece mais informações. Exceções serviriam para:

(1) permitir lidar com a falha eminente ou já ocorrida de uma operação.

(2) indicar as circunstâncias ou significado de um resultado válido.

(3) permitir ao invocador monitorar a operação.

Parece-nos importante relacionar o trabalho de Goodenough com a discussão que transcorria na época sobre programas estruturados e o uso de GOTOs. Na página 685, Goodenough diz textualmente: "... exceções e mecanismos de tratamento de exceção não são necessários apenas para lidar com erros. Eles são necessários, em geral, como um meio de intercalar convenientemente ações que pertencem a níveis diferentes de abstração." O mecanismo proposto por Goodenough pode ser visto dentro de um conjunto de propostas para substituição do GO-TO. Em [Knuth 74] temos um ótimo retrato desta discussão. A motivação principal para a existência de um comando de transferência é a eficiência. O caso (2) de Goodenough parece ser pensado para resolver os mesmos problemas que o comando de repetição com eventos sugerido por Knuth (ver exemplo dado para exceção ENDED na página 691 por Goodenough). O caso (c) sugere um tipo de estrutura semelhante aos iteradores posteriormente incluídos em algumas linguagens. O primeiro caso fica sendo

portanto o único que realmente trata de condições "excepcionais". Dentro deste caso Goodenough distingue dois subcasos como sendo de interesse especial: falhas de domínio e de imagem. Falhas de domínio são aquelas em que os parâmetros de um procedimento não satisfazem a assertiva de entrada; o exemplo citado é a ocorrência de uma letra em uma cadeia de dígitos. Falhas de imagem ocorrem quando uma operação não consegue satisfazer sua assertiva de saída; o exemplo citado é uma operação de leitura de um registro de um arquivo que se encontra na situação "fim de arquivo". A distinção entre estes dois casos parece um tanto artificial, uma vez que as falhas de imagem podem ser consideradas falhas de domínio, bastando para isto uma redefinição das assertivas (no caso da operação de leitura, a condição fim de arquivo poderia estar proibida pela assertiva de entrada). Temos então apenas a idéia geral de que uma exceção ocorre quando os operandos são tais que não é possível produzir um resultado adequado; além disto, de que existe uma necessidade de estruturas de transferência de controle eficientes que não têm nenhuma ligação intrínseca com condições de exceção.

II.2. Propostas estruturadas

Depois de Goodenough, vários autores se dedicaram ao problema de tratamento de exceções. Estes autores já demonstram, em geral, uma preocupação com os conceitos de programação estruturada, buscando propor mecanismos cuja interação com o restante da linguagem mantenha padrões de ortogonalidade e modularidade. A seguir discutiremos o conceito de exceção em alguns destes trabalhos.

A tese de Levin [Levin 77], dedicada à discussão do tratamento de exceções e à proposta de um novo mecanismo, admite explicitamente a falta de uma definição do conceito. Levin rejeita a visão de que uma exceção é um evento que ocorre raramente, e sugere que mecanismos de tratamento de exceção servem para tornar programas mais legíveis permitindo exibir com mais ênfase alguns casos (considerando-os "normais") e esconder outros (considerando-os "exceções"). Aqui surge uma questão que voltará a

ser levantada quando discutirmos o tratamento de exceções em linguagens com ênfase em definição de tipos abstratos de dados. Uma rotina é escrita por determinado programador em um contexto onde ele deseja dar ênfase ao caso A e esconder o caso B, e portanto coloca o caso B como exceção. Se mais tarde um outro programador necessita de uma rotina que faça exatamente a mesma coisa, porém num contexto onde o caso B não deve ser escondido e sim enfatizado, ele deverá escrever uma nova versão do procedimento ou tentar utilizar a existente apesar das possíveis consequências para a legibilidade de seu programa?

Em [Cristian 82] há uma tentativa de explicitar a definição de exceção. A idéia geral parece ser a mesma do caso (1) de Goodenough. Define-se uma relação binária **post** sobre o conjunto de estados internos possíveis que determina o serviço oferecido por um procedimento P. O par de estados (s', s) pertence a **post** se o estado s é um estado aceito como resultado de uma chamada de P no estado s'. O conjunto de estados s' tais que (s', s) $\in$ **post** para algum s é chamado o domínio padrão de P. Diz-se que uma exceção ocorre quando P é chamada em um estado que não pertence ao domínio padrão. Informalmente isto corresponde à idéia de que uma exceção ocorre quando uma rotina P é chamada numa situação em que nenhum estado final possível é aceitável. O exemplo dado neste trabalho torna a situação um pouco confusa. O procedimento P em questão retorna na variável interna **i** o valor da soma de **i** e **j**. Observa-se que, se P for implementada como "i := i + j" teremos um determinado domínio padrão (dos pares **i** e **j** cuja soma corresponde a um inteiro representável) mas que, por outro lado, se P for implementada erroneamente como "i := i*j" teremos como domínio padrão o conjunto, muito mais restrito, de valores para os quais **i*j=i+j**. Isto não parece absolutamente coerente com a definição anterior, que não é dada em termos da implementação de P (a menos que se entenda por "estado possível" um estado que possa ser produzido pela implementação corrente de P). No entanto, esta confusão entre especificação e implementação não invalida a formulação anterior.

Yemini publicou uma série de trabalhos sobre tratamento de exceções, incluindo sua tese de doutorado de 1980. Na seção da

tese entitulada "what are exceptions?" Yemini diz que, para obter máxima utilidade e confiabilidade, um programa deveria ser projetado para lidar com todos os tipos de entrada sem gerar erros de execução; dado que os sistema de tipos não são suficientemente poderosos para que um compilador possa garantir que as operações estão sendo chamadas com valores dentro do domínio esperado, a abordagem que ela propõe é que a assertiva de entrada de uma operação não suponha nada que não possa ser verif ado estaticamente, e que a própria operação identifique os estados onde a assertiva "verdadeira" não é satisfeita. Estes estados são denominados excepcionais. Esta descrição parece ficar na mesma linha que o enfoque de Cristian. Na mesma tese, na descrição dos objetivos de seu trabalho (pag. 7) Yemini reforça sua definição do conceito, afirmando que seu mecanismo pretende resolver este tipo de problema especificamente, em oposição ao de Levin, que pretende ser uma facilidade mais geral para comunicação e transferência de controle. No entanto, no mesmo trabalho é colocado que um mecanismo de tratamento de exceção deve prover todos os tipos de resposta já identificados na literatura (terminação, nova tentativa, retomada, etc). Estas duas afirmações parecem se chocar, pois não faz sentido que um novo mecanismo deva absorver características de outros que não foram projetados com o mesmo objetivo (a menos que os cinco tipos de resposta mencionados tenham sido identificados em mecanismos com os mesmos objetivos citados ?). Além disto, este critério para escolha de facilidades a serem oferecidas não parece o mais sensato. Se fosse adotado para o projeto de uma nova linguagem de programação o resultado certamente seria caótico.

Uma outra visão do conceito de exceção aparece em [Cheriton 86]. Neste trabalho se propõe que se defina uma exceção como qualquer evento que ocorre com freqüência baixa (definida por uma probabilidade de ocorrência). A idéia é que se otimize programas reescrevendo-os de forma a tornar mais rápida a execução dos casos "comuns" em detrimento dos casos "excepcionais". Esta visão nos remete mais uma vez ao trabalho de Knuth sobre GO-TOs. Apesar deste não conter nenhuma referência ao termo exceção, a preocupação básica envolvida nos dois casos é a eficiência. No

entanto, 12 anos separam estes dois trabalhos e neste intervalo de tempo parece-nos que houve uma mudança significativa no enfoque dado ao desenvolvimento de programas. O fato de os recursos computacionais terem se tornado menos escassos, o aumento na complexidade dos sistemas desenvolvidos e a pesquisa na área de otimização de código gerado estimulam que se pense cada vez menos em eficiência no momento em que se escreve um programa, e mais e mais em sua clareza e flexibilidade. Assim, vamos desconsiderar aqui mecanismos que se justifiquem basicamente pela eficiência, e vamos nos concentrar naqueles pensados com o objetivo de facilitar as tarefas de escrever e entender programas.

II.3. Implementações

A partir do final dos anos 70 muitas das linguagens propostas passaram a incluir mecanismos de tratamento de exceções. Nesta seção veremos o que os autores de algumas destas linguagens entendem por exceção. As linguagens escolhidas foram MESA, CLU, Ada e Eiffel por sua relativa difusão e disponibilidade de documentação.

A linguagem MESA oferece um mecanismo de tratamento de exceção extremamente poderoso, mas não foi possível encontrar nenhum trabalho que fornecesse informação sobre o conceito de exceção para seus autores. A única pista a este respeito é a descrição, contida no manual da linguagem, de duas propriedades do mecanismo. A primeira é que qualquer leitor do programa pode identificar imediatamente o fluxo normal e restringir sua atenção a ele; esta observação nos remete à motivação dada por Levin. A segunda propriedade apontada é o ganho de eficiência decorrente do uso do mecanismo proposto. No caso normal haveria redução do código executado, pois não é necessário testar valores a cada retorno de chamada de rotina. Nos casos excepcionais, o código necessário para implementar a busca do tratador associado seria maior do que o existente sem um mecanismo de exceção. Os autores argumentam que, dada a frequência relativa destes casos, a eficiência total aumenta com o uso do mecanismo.

CLU inclui um mecanismo de tratamento de exceção mais

restrito que o de MESA mas bastante poderoso. Em [Liskov 77] a motivação dada para este mecanismo é a confiabilidade que um programa deveria apresentar, descrita como sua capacidade de comportar-se de maneira "razoável" em um amplo espectro de situações. Mais uma vez fala-se de circunstâncias em que um procedimento pode terminar "normalmente" e outras, em que não há nenhuma ação possível, em que um outro procedimento deve ser notificado. A preferência pelo termo "exceção" em vez de "err " é justificada pelo fato de que a situação anormal pode não ser absolutamente errada, apenas menos desejável. Também reaparece a idéia de Goodenough de uso de exceções como um mecanismo mais geral de interação entre diferentes níveis de abstração. Na página 547 os autores dizem: "se o mecanismo de tratamento de exceção fosse eficiente o bastante exceções poderiam ser usadas para transmitir informações em situações normais e usuais". Na página 548 observa-se que todas as exceções que um procedimento pode sinalizar devem ser consideradas como parte daquela abstração procedural. Esta observação reforça a idéia de que exceções não são ocorrências realmente anormais, mas apenas resultados diferenciados: a interface de um procedimento define que ele pode terminar de várias maneiras, cada uma delas com retornos próprios. Uma destas maneiras, no entanto, ficará sempre caracterizada como "normal". Como já mencionado na discussão sobre o trabalho de Levin, isto prejudica a reutilização de código. É argumentável, no entanto, que este problema é inerente ao modelo de programação baseado em interfaces, uma vez que toda especificação de interface é feita pensando-se em aplicações específicas: a tarefa de construir uma interface que defina apenas a semântica da operação sem nenhuma indicação do significado dos parâmetros e resultados parece bastante complicada. Um exemplo pitoresco da particularidade impressa a interfaces pelo ponto de vista do autor encontra-se na descrição de tratamento de exceção dada numa resposta a uma questão de prova na graduação da UFRJ. O exemplo fornecido para a descrição é um procedimento que trabalha sobre um arquivo contendo uma árvore genealógica. Dado o nome de uma pessoa, este procedimento retorna o nome de seu pai e de sua mãe. A exceção ocorreria quando o nome dado como entrada correspondesse

a uma pessoa sem pai. Segundo a autora do exemplo, o tratamento desta exceção refletiria o conservadorismo maior ou menor do usuário do procedimento. No entanto, pode-se dizer que a própria especificação deste procedimento já imprime algum julgamento moral ao considerar "normal" que uma pessoa tenha pai e mãe.

Na definição de Ada não é nada claro o que se entende por exceção. O rationale [Ichbiach 79] diz que o mecanismo de tratamento de exceção provê uma facilidade para a terminação local quando da ocorrência de um erro, e coloca a proposta para tratamento de exceções de Ada na família das propostas que "restringem exceções a eventos que podem ser considerados (em algum sentido) erros". Esta família de propostas é colocada em oposição a outra, onde são classificados os trabalhos de Levin e de Goodenough, que "considera tratamento de exceções como uma técnica de programação normal para eventos que são raros mas não necessariamente erros". O próprio rationale diz que o conceito de erro é subjetivo e não se propõe a explicitá-lo.

A linguagem Eiffel [Meyer 88a] segue o paradigma de orientação a objetos: Eiffel permite que assertivas lógicas sejam verificadas em pontos arbitrários de seus programas. Na discussão da linguagem apresentada em [Meyer 88b] encontramos uma das poucas definições do termo exceção. Na página 149 deste trabalho, o autor diz que uma exceção pode ocorrer durante a execução de uma rotina r como resultados de uma entre 7 situações descritas a seguir.

(1) A pré-condição de r é falsa na entrada.

(2) A pós-condição de r é falsa na saída.

(3) A invariante de uma classe é violada na entrada ou saída de r.

(4) Uma violação de uma outra assertiva qualquer ocorre durante a execução de r.

(5) Uma rotina chamada por r falha (não consegue cumprir sua tarefa).

(6) r tenta executar uma método associado a uma variável a em um instante em que a não tem valor.

(7) Uma operação executada por r resulta em uma condição anormal detectada pelo hardware ou pelo sistema operacional.

O conceito de falha, usado na descrição da situação 5, é definido

recursivamente como sendo o resultado de uma rotina que termina através de uma exceção.

II.4. Classificação de estados excepcionais

Podemos, a partir da discussão anterior, identificar alguns contextos em que os trabalhos acima julgaram ser adequado utilizar o conceito de exceção. O primeiro é aquele em que, dada uma operação e um estado inicial, nenhum estado final é admissível para a continuação sequencial do programa (encaixa-se aqui a definição de exceção de Cristian). Dentro dele, podemos distinguir o que chamaremos genericamente de "overflows", ie, casos em que é impossível para a implementação representar totalmente a abstração (conforme Black), e os que chamaremos de funções parciais, onde as operações são definidas apenas para um subconjunto dos valores possíveis para seus parâmetros. Como exemplo do primeiro tipo podemos citar falta de memória no momento de realizar uma alocação, e como exemplo do segundo uma divisão por zero. O segundo contexto que aparece repetidamente é aquele em que uma operação pode terminar com vários resultados, nenhum deles necessariamente errados (CLU, etc). Como exemplo clássico temos uma rotina que verifica se um identificador está em uma tabela de símbolos. Neste contexto, encontramos repetidamente a idéia de que a colocação de uma das situações como normal (em geral a mais freqüente) e das outras como exceções aumenta a legibilidade do programa (Levin, Mitchell, etc). Obviamente a fronteira entre os casos identificados acima não é bem definida. Uma operação de leitura que encontra fim de arquivo é classificada por Goodenough como exemplo de estado final inadmissível, enquanto aqui tenderíamos a encaixá-la como uma operação com mais de um estado final possível. No entanto, esta divisão parece refletir bastante bem as principais correntes, e parece ser a mesma classificação comentada no rationale de Ada.

Um terceiro tipo de situação descrita em vários trabalhos é aquela em que se sente necessidade de uma estrutura de controle poderosa para gerenciar "entrelaçamento de níveis" (transferência de controle/comunicação). Este contexto só é mencionado como ponto

de partida no trabalho de Goodenough e realmente não constitui uma situação conceitual como as duas primeiras. No entanto vários trabalhos se preocupam com ele, em especial o de Levin que, com a idéia de exceções ligadas a estruturas (que será discutida na seção IV) fornece um mecanismo de transferência de controle especialmente poderoso.

III. Alternativas

III.1. Exceções como erros fatais

Em relação ao primeiro contexto identificado acima, onde nenhum estado final é possível para uma operação, as alternativas parecem ser: (a) definir assertivas de entrada mais rigorosas e supor que elas sãc sempre honradas na chamada da operação, ou seja, obrigar o programador a evitar que tais situações ocorram; (b) definir mecanismos na linguagem que permitam evitar automaticamente a ocorrência destas situações. Uma forma para a segunda solução é a definição de um sistema de tipos rigoroso o suficiente para garantir a pertinência de valores a domínios arbitrários.

Esta parece ser a idéia de Yemini quando diz que mecanismos de tratamento de exceção são necessários porque não existem compiladores suficientemente poderosos para realização das verificações estáticas necessárias. No entanto, é difícil imaginar como uma situação do tipo overflow em uma operação de soma poderia ser verificada estaticamente; cada um dos valores pode ser resultado de uma série de condições desconhecidas em tempo de compilação. É interessante relacionar a observação de Yemini com o trabalho da mesma autora com a linguagem NIL [Strom 86], onde é proposto um mecanismo linguístico chamado "typestate", inspirado nos conceitos descritos por Dijkstra em [Dijkstra 76]. Um tipo é associado a um conjunto de estados possíveis sobre os quais existe uma ordenação parcial. Em cada ponto do programa uma variável está em um estado determinado. Se o compilador não pode estabelecer o estado exato, o que ocorre no caso de existir mais de um caminho possível até este ponto do programa, um limite inferior de estado é calculado. Toda operação aplicada a qualquer objeto tem que ser aplicável ao estado correspondente a este limite inferior. Os autores não pretendem que este mecanismo suprima a necessidade de facilidades de tratamento de exceção, pois mencionam a existência de tais facilidades na linguagem; no entanto, vale a pena discutir a relação entre estes mecanismos, uma vez que o conceito de "typestate" representa uma passo na direção apontada por Yemini de

maior rigor na verificação estática. A noção de "typestate" permite que o compilador proíba acessos a variáveis não inicializadas (o que é bastante interessante) mas não lhe possibilita a verificação de domínios. Isto requeriria a redefinição de estados para cada operação definida com restrições sobre a entrada diferentes de restrições anteriores.

Um outro trabalho nesta direção é descrito em [Eggert 81]. A idéia é propor uma extensão de Pascal onde o maior número possível de erros de execução são transformados em erros de compilação. Uma série de imposições são colocadas; por exemplo, todas as variáveis devem ser inicializadas no momento em que são declaradas. Tipos são divididos em conjuntos de subtipos que correspondem de certa forma aos estados da proposta de Yemini, mas aqui o programador é responsável por declarar as variáveis como sendo de determinado subtipo. Isto gera uma série de dificuldades: por exemplo, um tipo ponteiro é sempre dividido em um subtipo "ponteiro válido" e um subtipo "NIL", e uma variável declarada como "ponteiro válido" não pode nunca ter valor NIL. Além da mesma impossibilidade de verificação de domínios mencionada na proposta de typestate, a programação torna-se extremamente complexa.

III.2. Exceções como mecanismos multi-saída

No segundo contexto identificado na seção II.4 temos o problema já discutido da falta de simetria imposta pelo uso de um mecanismo de TE. Uma saída para evitar este problema mantendo o mesmo "poder de expressão" seria prover uma facilidade multi-saída. A idéia de uma estrutura multi-saída não é nova. Em [Zahn 74] discute-se um comando de repetição que permitiria realizar transferências de controle de maneira estruturada, permitindo a resolução de problemas criados pela eliminação do GO-TO. A forma geral deste comando é:

```
until Ev1 or Ev2 or ... or Evn
  S0
then case
  Ev1: S1
  Ev2: S2
  ...
  Evn: Sn
```

Zahn cita quatro situações em que a falta de GO-TO torna programas menos legíveis e eficientes, e onde, portanto, o comando acima seria útil:

(1) terminação múltipla de uma repetição
(2) terminação de repetições aninhadas
(3) árvores de decisão com nós duplicados
(4) tratamento de erro multi-nível

Segundo ele, o comando proposto resolve problemas nas quatro classes. No entanto nos exemplos dados os eventos são sempre tratados localmente: o "multi-nível" mencionado na situação 4 refere-se aparentemente a níveis de aninhamento dentro do corpo de um procedimento, e não a níveis de abstração. Na página 178 ele dedica um parágrafo a "eventos em procedimentos" e diz que a utilidade do mecanismo para tratamento de erros sugere que ele seja usado também para sinalizar eventos a serem tratados em outra unidade de compilação, mas não se aprofunda sobre este tipo de extensão. Como se garantiria que um procedimento chamado sinalizasse um dos eventos declarados? Uma sinalização faria a cadeia dinâmica ser percorrida até que se encontrasse um comando de repetição englobante?

O próprio Knuth [Knuth 74] fala de tratamento de erro como uma das situações em que o GO-TO se faz necessário, mencionando explicitamente a necessidade de transferir o controle para um procedimento vários níveis acima do corrente na cadeia dinâmica, mas não discute a utilidade do comando de Zahn para este caso.

O que nos parece é que o comando proposto resolve bem as situações (1) e (2). A situação (3) não parece realmente precisar de um mecanismo especial, pois pode perfeitamente ser tratada com a introdução de chamadas de procedimento em substituição aos nós repetidos. Além disto esta situação não envolve nenhuma repetição: a utilização do comando com repetições para solucioná-la implica na aplicação sistemática de um comando de repetição em circunstâncias onde já se sabe previamente que este será executado exatamente uma vez. A situação (4), com a a interpretação mencionada acima (multi-nível referindo-se a níveis de aninhamento), funde-se com a situação (2).

Black propõe que se utilizem facilidades já existentes nas linguagens para definir procedimentos com várias saídas simétricas. Black utiliza o construtor de tipos ONEOF, semelhante às uniões de C ou registros variantes de Pascal, porém com controle estático mais rigoroso. O compilador exige conversões explícitas do tipo oneof para os tipos "componentes", e estas geram erro de execução caso o valor em questão não tenha sido gerado com a conversão contrária (do mesmo tipo componente para o tipo construído com oneof).

Os mecanismos propostos por Zahn e Black expressam a mesma idéia, porém de formas sintaticamente tão diferente que a relação entre eles se torna pouco óbvia. É interessante notar que CLU introduz um comando **exit** para adaptar seu mecanismo de controle de exceções para a solução do problema da multi-saída local (o comando **exit** é explicitamente inspirado na proposta de Zahn). Parece-nos vantajoso que os mecanismos multi-saída guardem a mesma forma para saídas locais ou multi-nível. Uma forma de fazer isto seria definir uma sintaxe para declaração de procedimentos com os retornos possíveis sendo indicados por nomes de eventos, como é feito nos mecanismos de exceção de CLU, Yemini e outros, e não pelo tipo de retorno, introduzindo a simetria de exigir que todos os casos exibissem um tal "rótulo". Isto é, o cabeçalho de um procedimento seria semelhante ao de CLU mas sem uma sintaxe diferenciada dedicada ao retorno do caso "normal". Sobre este ponto, porém, deve ser feita a seguinte observação. A rotina de procura de um nome em uma tabela é sempre o exemplo dado de procedimento com mais de uma saída "normal" possível, onde a normalidade de encontrar ou não o nome depende do contexto que o chama. Não nos parece que existam realmente muitos casos deste tipo; em geral, ao escrever uma interface (que, como já mencionado, inclui em geral algum "pré-conceito"), uma das saídas possíveis é destaca como a normal.

A solução de Black, além de não ser facilmente identificada como uma extensão do mecanismo multi-saída local, tem a desvantagem de causar uma certa poluição do programa com comandos de conversão. A motivação principal para ela é a simplicidade da

linguagem: Black argumenta que é possível construir estruturas de controle semelhantes àquelas providas por mecanismos de controle de exceção sem introduzir novos conceitos na linguagem. Segundo Black, uma linguagem não deveria oferecer mais que uma forma de expressar uma mesma idéia (pag. 170), pois isto obriga o usuário a se preocupar com as implementações para poder escolher entre elas. No entanto, levando-se este raciocínio ao extremo, voltaríamos a programar sempre na linguagem da máquina, uma vez que ela é suficiente para expressar qualquer idéia de uma linguagem de alto nível. O fato da linguagem oferecer mais de uma construção para executar a mesma tarefa pode permitir ao usuário escolher aquela que ele considera mais clara. Parece-nos que a observação de Black de que isto estimula o usuário a pensar em implementações não procede, uma vez que uma boa linguagem aliada a um bom compilador pode liberá-lo deste tipo de preocupação.

III.3. Exceções como estruturas de controle

Os dois primeiros cenários podem ser unificados com a ótica de que são duas situações onde é necessário alertar o usuário sobre condições de terminação distintas , sejam elas catastróficas ou não. A situação em que se deseja uma interação maior entre diferentes níveis de abstração parece-nos suficientemente diferente das duas anteriores para que não se espere solucionar as três com um mesmo mecanismo (o que as três têm em comum é a necessidade de uma estrutura de transferência controle poderosa, mas isto não implica que a mesma estrutura seja indicada em todas).

O exemplo dado em [Goodenough 75, pag 690] sugere, como já mencionado, o uso de uma facilidade como os iteradores de CLU. Em outras linguagens que não oferecem este conceito, o mesmo efeito pode ser obtido através do uso de co-rotinas. Em qualquer uma destas soluções a própria sintaxe parece espelhar melhor a natureza do problema do que o uso de uma sinalização a cada iteração.

Outros casos onde tal interação é necessária podem ser resolvidos através do uso do mecanismo clássico de passagem de

IV. Propostas existentes de mecanismos de tratamento

Nesta seção examinaremos algumas propostas existentes para mecanismos de tratamento de exceção; as propostas selecionadas para discussão são aquelas em que os autores apresentam preocupação explícita com este tópico e onde se considerou que o mecanismo fornecido apresentava alguma inovação significativa (isto significa que não vamos discutir, por exemplo, linguagens onde os autores, por considerarem importante a existência de mecanismos de tratamento de exceção mas não terem interesse específico neste tópico, simplesmente agregaram um mecanismo já proposto anteriormente).

Nosso objetivo é classificar o comportamento destes mecanismos em relação a alguns pontos, a saber:

(a) declaração de exceções
Como são definidas as exceções que podem ser levantadas em cada contexto?

(b) associação de uma exceção a um tratador
Como se dá a procura de um tratador no momento em que a exceção é gerada? Normalmente esta procura é feita através da cadeia dinâmica de ativações. Alguns autores sustentam que devem ser percorridos o número de registros de ativação que for necessário até encontrar um tratador; esta estratégia é chamada propagação automática. Outros consideram que a rotina responsável pela chamada do sinalizador deve definir um tratador.

(c) fluxo de controle após execução do tratador
Este ítem envolve dois pontos. O primeiro diz respeito aos modelos possíveis de retomada e terminação. No modelo de retomada, quando uma exceção é sinalizada um determinado tratador é excutado e a seguir o controle retorna para o comando seguinte ao que sinalizou. No caso do modelo de terminação, a rotina que contém o comando sinalizador é terminada e, após execução do tratador, o controle vai para a rotina que a chamou. Neste caso, a questão que se coloca é: para onde vai o controle após execução do tratador?

procedimentos como parâmetros. Numa rotina de alocação de memória, por exemplo, se a idéia é pedir para outro nível de ativação liberar memória para que seja possível realizar a alocação pedida, isto pode ser feito através da chamada de um procedimento **CriseEspaço** passado como parâmetro para **Aloca**.

IV.1 Goodenough

O mecanismo proposto por Goodenough já foi amplamente criticado por sua complexidade, e aparentemente nunca chegou a ser incorporado a nenhuma linguagem de programação. No entanto, por se tratar do trabalho pioneiro nesta área, achamos que uma discussão sobre propostas existentes não pode deixar de abordar este trabalho.

A notação de Goodenough exige descrição explícita das condições que uma rotina pode sinalizar em seu cabeçalho. Um tratador é associado a uma unidade sintática contendo a ativação da operação que pode gerar a exceção (expressão, comando, ou grupo de comandos em um bloco). O alcance de um tratador é definido como sendo esta unidade sintática a menos de unidades sintáticas mais internas que tenham uma definição de tratador para a mesma exceção. Um tratador é ativado se a exceção para a qual ele é definido é sinalizada por uma ativação dentro de seu alcance. Por exemplo, no trecho de programa:

```
1    DO;
2       H = F(A) + F(B)[X: H1];
3       I = (F(C) + F(D))[X: H2];
4       J = F(E);
5    END; [X: H3]
```

a chamada F(B) está no alcance do tratador H1, enquanto as chamadas F(C) e F(D) da linha 3 estão no alcance do tratador H2. A chamada F(A) na linha 2 e a chamada F(E) na linha 4 está no alcance do tratador H3 definido na linha 5. Este será o tratador chamado caso X seja sinalizada durante uma destas chamadas de F. Esta facilidade de associar um tratador a vários pontos de ativação por um lado evita que o programador tenha que repetir diversas vezes uma definição do tratador mas por outro lado facilita a ocorrência de erros. O programador pode esquecer de definir um tratador para determinada ativação e, se existir uma definição mais externa uma sinalização nesta ativação acionará um tratador possivelmente diferente do desejado. No exemplo dado, todas as ativações consideradas eram da mesma rotina, mas tendo em mente que uma mesma exceção X pode ser sinalizada por diferentes

rotinas F e G o problema se torna mais grave, uma vez que o programador pode simplesmente se esquecer de que G sinaliza X e não definir um tratador para a deteção de X por G; esta deteção causará a execução do tratador definido para a sinalização de X por F, erro que pode ser difícil de descobrir. Goodenough menciona este problema mas argumenta que o conforto provido por esta facilidade é compensação suficiente para ele. Esta notação para associação de sinalizações a tratadores será uma constante em trabalhos posteriores.

Uma ativação de rotina que sinaliza uma exceção E fora do alcance de qualquer tratador de E é considerada uma erro de compilação, a menos que E seja uma exceção pré-definida (neste caso, se E for sinalizada por esta ativação, será chamado um tratador pré-definido).

Uma exceção nesta notação é apenas um nome: parâmetros não são permitidos. Os comandos dentro de um tratador não estão no seu próprio alcance, e sim no alcance de tratadores definidos mais externamente.

Três tipos de exceção são definidos; este tipo deve ser especificado no cabeçalho do procedimento que sinaliza a exceção, e pode ser:

(a) ESCAPE - a terminação da exceção é exigida;
(b) NOTIFY - a retomada da operação é exigida;
(c) SIGNAL - permite-se retomada ou terminação; a escolha é feita pelo tratador.

Exceções também podem ser tratadas localmente. Neste caso, não é necessário que elas sejam declaradas. Uma construção do tipo

```
DO WHILE ()
  DO WHILE ()
    ....
    IF () THEN
      ESCAPE X
    ELSE
      ....
    ....
  END;
END; [X: ...]
```

faz com que os dois loops sejam interrompidas no caso de X ser

sinalizada. Um comando do tipo **ESCAPE** X, **NOTIFY** X ou **SIGNAL** X, onde X não é declarada no cabeçalho do procedimento, deve estar necessariamente no alcance de algum tratador de X.

O controle do fluxo de execução está fortemente associado à posição física dos tratadores. Sempre que um tratador é finalizado com a terminação da rotina sinalizadora, o controle passa para o comando seguinte ao bloco ao qual este tratador está associado.

Goodenough faz ainda uma série de definições de comandos para lidar com casos particulares que ele considera suficientemente frequentes para merecerem facilidades especiais. Este excesso de comandos, que torna o mecanismo bastante complicado, já foi suficientemente criticado (ver por exemplo [Cristian 79] pg 3.6) para não merecer maior discussão aqui.

IV.2. Levin

O trabalho de Levin propõe um mecanismo a ser incorporado à linguagem de programação Alphard [Wulf 76]]; aparentemente esta extensão não chegou a ser implementada, pelo menos até a redação da tese.

Nesta proposta, uma condição tem que ser declarada na seção de especificações de uma forma de Alphard, que constitui a unidade de compilação da linguagem (correspondente ao módulo de Modula-2). Dentro da forma, os procedimentos que sinalizam esta condição devem informar que o fazem em seu cabeçalho.

Um ponto bastante diferente neste trabalho é a definição de dois tipos de condição: condição de estrutura e condição de fluxo. Uma condição de estrutura se aplica a uma estrutura de dados, e afeta o estado interno do módulo que a sinaliza. Uma condição de fluxo se aplica apenas à chamada de procedimento que o sinaliza. Levin fornece o seguinte exemplo. Uma função de gravação de arquivo poderia sinalizar duas condições: "arquivo inconsistente" e "arquivo ler somente". A primeira descreve um problema estrutural, uma inconsistência na implementação de uma arquivo (comentar mais tarde que tal erro deveria ser detectado no momento que o arquivo foi aberto?). A segunda descreve a impossibilidade

de realizar a operação de gravação em um arquivo ler somente, que é um problema de fluxo. A declaração de uma condição não indica a qual das duas categorias ela pertence. O cabeçalho de um procedimento que a sinaliza é que indica a que objeto a condição de refere. No exemplo acima, teríamos o cabeçalho:

```
function file_write (f: file)
  raises file_inconsistent on f
  raises file_read_only on file_write
  ...
```

Esta distinção se reflete na escolha do tratador a ser executado no momento em que uma condição é sinalizada. Esta escolha ocorre em duas fases: na primeira determina-se qual o conjunto de tratadores elegíveis, e na segunda, usando-se uma política de seleção, um subconjunto deste conjunto é escolhido (o mecanismo permite a invocação de mais de um tratador).

Um tratador T é considerado elegível para tratar uma condição C sobre uma instância I se ele é associado com um contexto de programa onde I é visível (tratadores mais internos "desabilitam" tratadores mais externos). Esta definição torna trivial o caso das condições de fluxo, pois o único contexto onde uma ativação de rotina é visível é aquele onde se executa a chamada. A situação é bem mais complicada para condições de estrutura, pois uma mesma variável pode ser visível a partir de vários módulos (o exemplo clássico é um pool de memória usado por vários módulos para pedidos de alocação).

A cada condição é associada uma política de seleção. Uma política possível é a invocação de todos os tratadores envolvidos. Uma outra política possível é a invocação de tratadores até que a condição sinalizada deixe de ser verdadeira No exemplo do banco de memória, se a condição **memória insuficiente** fosse detectada durante uma tentativa de atender um pedido de alocação, a primeira política corresponderia à decisão de que neste momento é aconselhável que todos os módulos usuários do banco liberem a memória que não estiverem usando, enquanto a segunda corresponderia à decisão de fazer os módulos usuários liberarem memória até que haja memória suficiente para satisfazer o pedido gerador da sinalização. Para Levin, uma implementação de seu

mecanismo proveria um conjunto de políticas pré-definidas e um conjunto de primitivas que permitiriam a definição de outras políticas.

Após a execução do conjunto de tratadores selecionado o controle volta para o sinalizador (modelo de retomada). Levin argumenta que se um tratador pudesse terminar a rotina sinalizadora, poderia tornar-se impossível manter as abstrações do módulo que a contém. Este argumento pode fazer sentido num confronto com um mecanismo como o de Goodenough, onde em certas situações o controle pode ou não voltar ao sinalizador. No entanto, se este sabe que a ação de sinalização é sua ação final, não parece haver problemas em tomar as mediads necessárias para deixar o módulo em estado coerente antes de realizar esta ação.

Uma peculiaridade interessante é que, apesar do controle sempre retornar ao sinalizador após a execução do tratador, este pode especificar um alteração no fluxo de controle no contexto ao qual está associado (contexto de ativação onde foi feita a chamada do sinalizador, no caso de condições de fluxo). Um tratador pode especificar, por exemplo, que quando o sinalizador terminar execução, o contrôle deve retornar não para o comando seguinte à chamada, e sim para o final do bloco.

Na seção "simplificando o uso do mecanismo", Levin afirma que em casos de terminações anormais (como overflow) é desconfortável para o programador ser forçado a usar um comando **return** explícito após a sinalização da condição; sua sugestão é que se defina uma macro, em cima das primitivas oferecidas, que para o programador se comporte como uma sinalização com terminação. Levin observa que tal definição simplifica o entendimento sem dificultar verificabilidade.

A complexidade do mecanismo de Levin é o fator de maior peso contra ele. A tarefa de entender como são associados tratadores a sinalizações é árdua, e com um ponto perigoso. Uma linguagem pode se dar ao luxo de oferecer um mecanismo poderoso porém complexo se o compilador é capaz de garantir o bom uso deste mecanismo (==>como no caso de passagem de procedimentos como parâmetros? ==> rediscutir este parágrafo). No entanto, a introdução de um

mecanismo com estas características se torna muito mais discutível quando é possível que o programador escreva um programa acreditando em determinado comportamento e, não recebendo nenhum aviso do compilador, venha a deparar-se com um comportamento diferente.

IV.3. CLU

Uma das poucas implementações existentes de um mecanismo de tratamento de exceções encontra-se na linguagem CLU.

Exceções são declaradas no cabeçalho das rotinas que as sinalizam; esta declaração inclui possíveis parâmetros. Este é o primeiro mecanismo que discutimos que permite parametrizacão do tratador. Sem a capacidade de passar parâmetros tanto o número de variáveis globais aumenta, dimminuindo a força das abstrações, como também aumenta o número de tratadores que devem ser definidos. Parece-nos bastante óbvio que a introdução da parametrização aumenta em muito a expressividade do mecanismo.

A associação com o tratador é feita através da colocação de uma cláusula **when** em um bloco que contenha a invocação da rotina sinalizadora (como em Levin). Se determinada invocação de uma rotina B sinaliza uma exceção E e no corpo da rotina que chamou B não existe nenhuma associação de tratador para E válida para a chamada em questão, esta última sinaliza a exceção pré-definida **failure**. Esta é uma forma de oferecer uma certa flexibilidade em relação a propagação automática sem incorrer no perigo de quebra de abstração (o chamador de A não tem porque entender o significado da exceção E).

A execução de um tratador de E em CLU sempre implica na terminação da rotina sinalizadora. Os autores da proposta argumentam que o modelo de retomada cria interdependências desconfortáveis entre módulos em níveis diferentes de abstração, dado que o sinalizador tem que ter uma certa consciência das ações do tratador; a posição adotada por eles é que a flexibilidade que se ganha ao permitir a retomada não compensa a perda de simplicidade associada. Após a execução do tratador o controle

passa para o comando seguinte ao bloco ao qual o tratador está associado, como na proposta de Goodenough.

A cláusula **when** também pode ser usada para fazer associações com sinalizações locais; esta facilidade é baseada na proposta de Zahn (discutida na seção ??). A sinalização deve ser feita através do comando **exit**, ao invés de **signal**, para que o tratador seja procurado localmente. Neste caso as exceções não são declaradas antes de seu uso nas cláusulas **exit** e **when**.

IV.4. Yemini

Yemini e Berry [Yemini 85] definem um mecanismo chamado por eles de "modelo de substituição". O trabalho propõe que este mecanismo seja incorporado a uma extensão de Algol 68 que inclua pacotes como os de Ada.

As exceções são declaradas no cabeçalho dos procedimentos que as sinalizam. Esta declaração inclui a declaração de uma lista de parâmetros a ser passada no momento da sinalização e dos possíveis valores retornados.

Um tratador é associado com uma invocação através de um cláusula **on** ligada a algum bloco que contenna esta invocação estaticamente. Neste ponto esta proposta se distingue das anteriores, pois este aninhamento estático pode ultrapassar as fronteiras de definição de procedimentos. Este enfoque permite que tratadores pré-definidos dejam considerados como associações feitas num bloco padrão que engloba todos os programas de usuário. Como nos mecanismos anteriores, uma cláusula **on** faz com que o tratador definido se sobreponha a outros associados a blocos mais externos. Uma diferença para os mecanismos já vistos é que a posição do tratador ativado quando ocorre uma sinalização não tem influência alguma sobre o fluxo de controle após o término do tratador.

O modelo não permite propagação automática de exceções. Logo, dada a chamada de uma rotina que sinaliza uma exceção, deve existir uma cláusula **on** para esta exceção em um bloco que contenha esta chamada estaticamente, ou o programa será considerado errado.

Contra a propagação automática de exceções, Yemini argumenta que ela implica numa quebra de abstração. Como já mencionado na discussão sobre CLU, se uma rotina A chama uma rotina B, A não deve sequer saber que rotinas B chama, e muito menos que exceções podem ser sinalizadas por elas.

Já foi mencionado na seção II que Yemini considera que um mecanismo de tratamento de exceções deve prover todos os tipos de comportamento identificados na literatura. Dentro desta visão, os seguintes tipos de resposta são previstos:

(a) retomada do sinalizador
(b) terminação do sinalizador
(c) nova tentativa de chamar o sinalizador
(d) propagação da exceção
(e) transferência de controle

O mecanismo proposto oferece, estritamente falando, dois tipos de resposta, que podem ser usados para contruir os cinco tipos acima. As duas possibilidades são:

(a) retomar o sinalizador, substituindo a expressão sinalizadora;
(b) terminar o sinalizador, substituindo a chamada.

Cabe observar aqui que o mecanismo proposto e a linguagem hospedeira são orientados a expressões. Esta opção é justificada pela exigência de ortogonalidade do mecanismo de tratamento de exceções em relação ao resto da linguagem. Se uma chamada qualquer de rotina pode sinalizar uma exceção, em particular uma chamada dentro de uma expressão poderá fazê-lo, e isto implica que uma expressão poderá ter efeitos colaterais (==> comentar?) gerados pelas ações de um tratador. Com o enfoque de orientação a expressões o tratamento de exceções em expressões e comandos assume uma forma única. Também por motivos de uniformidade o mecanismo permite que qualquer bloco da linguagem sinalize uma exceção, o que significa que uma exceção pode ser tratada na mesma rotina que a sinalizou.

A terminação normal do tratador implica na retomada da rotina sinalizadora; neste caso a expressão retornada pelo tratador

substitui o valor que seria retornado pela expressão sinalizadora.

Como já dito acima, a declaração de uma exceção no cabeçalho ou bloco sinalizador inclui a declaração dos possíveis valores retornados. Estes possíveis valores correspondem aos dois tipos de retorno, retomada e terminação.

Uma característica importantes desta proposta é a possibilidade de uma verificação estática completa. Como não existe propagação automática de exceções, o compilador pode verificar se existe um tratador definido para cada exceção que pode ser sinalizada, e se o tratador está retornando tipos compatíveis com os exigidos pela declaração da rotina sinalizadora.

No caso do tratador optar pela terminação da rotina sinalizadora, o controle de execução passa para o comando seguinte à chamada da rotina sinalizadora. Esta escolha contrasta com as vistas anteriormente, onde o controle passa para o comando seguinte ao bloco ao qual o tratador está associado.Caso se deseje um comportamento deste tipo, pode-se usar a facilidade de sinalização de exceções a nível de bloco. Um tratador pode, como última ação, sinalizar uma exceção declarada em um bloco mais externo, e o tratador desta nova exceção pode terminar este bloco.

IV.5. MESA

Os projetistas de MESA parecem compartilhar com Yemini a noção de que todos os tipos de resposta já trabalhados devem ser oferecidos num mecanismo de tratamento de exceção, mas não sua exigência de simplicidade e ortogonalidade. Com isto, a linguagem oferece um mecanismo poderoso porém bastante complexo.

MESA define o construtor **SIGNAL** para declaração de tipos de exceções; este construtor é análogo ao usado para definir tipos de procedimentos em linguagens como Pascal, permitindo que se declare os parâmetros e o valor retornado por um tipo de exceção. A linguagem permite que estes tipos sejam usados inclusive para declaração de variáveis.

O fato das exceções serem tipadas é interessante uma vez que

resolve um problema existente nas soluções de CLU e de Yemini. Nestes mecanismos, se um tratador para uma exceção E1 é associado a um bloco que contém invocações de rotinas R1 e R2 tais que ambas sinalizam E1, o tratador T1 será ativado em caso de ocorrência de E1 qualquer uma das duas invocações, se os parâmetros forem compatíveis, uma vez que a associação é feita só pelo nome da exceção. Este pode não ser o efeito desejado pelo programador, como mencionado na descrição do mecanismo de Goodenough. Se o cabeçalho de cada rotina pudesse declarar as exceções que sinaliza usando nomes de tipos que seriam exportados pela mesma interface que exportasse a rotina, a situação discutida acima só seria aceita pelo compilador se as duas rotinas declarassem E1 como sendo do mesmo tipo, o que seria uma garantia a mais para o programador.

Por outro lado, o fato dos tratadores de exceção poderem ser variáveis ou mesmo parâmetros acarreta problemas, como o entendimento do escopo em que serão executados, que indiscutivelmente tornam o uso do mecanismo mais complexo.

A associação de uma sinalização com um tratador segue novamente o princípio do tratador ligado ao bloco mais interno, mas a diferença em relação aos mecanismos já vistos é o escopo da procura. A procura do tratador começa do corpo da rotina onde ocorreu a sinalização, passa, caso não tenha sucesso, para o corpo da rotina que a chamou, e prossegue pela cadeia dinâmica caso aí também não seja encontrado um tratador. Ou seja, uma exceção é propagada automaticamente pela cadeia dinâmica até que seja encontrado um tratador para ela ou o tratador genérico **any** (Caso a cadeia dinâmica seja esgotada sem que um tratador seja encontrado, o depurador assume o controle da excução).

Este mecanismo de associação não parece compatível com a ênfase em interfaces dada pela linguagem. O próprio manual da linguagem diz, na pag. 139: "...como sinais podem ser propagados diretamente pela hierarquia de chamadas, o programador deve considerar o tratamento não apenas de sinais gerados diretamente dentro de qualquer procedimento chamado, mas também quaisquer sinais gerados indiretamente como resultado de tal chamada". Este

tipo de preocupação não parece absolutamente razoável, dada a argumentação já apresentada na discussão do mecanismo de Yemini.

Um tratador pode, como na proposta de Yemini, terminar pedindo terminação ou retomada da rotina sinalizadora, ou ainda rejeitar o sinal. A rejeição de um sinal implica na continuação do processo de busca de um tratador, como se nenhum tivesse ainda sido encontrado. A retomada funciona como nos modelos de Yemini e de Levin. A terminação ocorre se o tratador executa um comando de transferência de controle; este pode ser um comando como **goto** ou **exit** ou um entre dois comandos específicos do mecanismo de tratamento de exceções, **continue** ou **retry**. O comando **continue** faz com que o controle passe, como em CLU, ao comando seguinte ao bloco ao qual o tratador está associado. O comando **retry** faz com que seja executado novamente o comando que gerou a sinalização.

Qualquer que seja a forma que leve à terminação da rotina sinalizadora, o mecanismo de tratamento de exceções passa para cada registro de ativação, desde o registro onde ocorreu a sinalização até chegar ao registro contendo o tratador, a exceção especial **unwind**. Esta exceção indica que a estrutura vai ser terminada e a intenção é permitir que cada ativação tome providências finalizadoras para que seu estado fique coerente. Os tratadores de **unwind** são escritos explicitamente pelo programador de cada rotina.

A necessidade desta solução bastante complexa para o problemas de garantir estados internos coerentes é mais uma decorrência da estratégia de propagação automática. A questão é que, com esta estratégia, um procedimento P qualquer tem que tratar cada chamada de procedimento como uma possível saída definitiva (cf. [Cristian 79] pag. 3.10), uma vez que o procedimento chamado pode, direta ou indiretamente, sinalizar uma exceção para a qual só seja encontrado um tratador num registro de ativação anterior ao de P na pilha e que opte pela terminação.

IV.6. Cristian

A tese de Cristian descreve uma proposta de tratamento de

exceção bastante semelhante à de CLU. O trabalho se destaca pelo cuidado e profundidade com que cada uma das opções é discutida (terminação x retomada, propagação automática, etc). Esta proposta inclui toda uma discussão voltada para um problema que não abordamos aqui, de tolerância a falhas. Nesta direção, o trabalho propõe uma facilidade interessante: uma primitiva **reset**, a ser executada antes de sinalizações. O uso desta primitiva seria restrito a procedimentos definidos na interface de algum módulo (exportados), e implica no restabelecimento do estado interno "em vigor" no momento em que foi chamado o procedimento, ou seja, o estado inicial na invocação corrente.

IV.7. Ada

A proposta de Ada não inclui nenhuma novidade em relação à de CLU ou a de Yemini, a não ser a interação dos mecanismos de tratamento de exceção com os de programação concorrente: esta interação torna o mecanismo extremamente difícil de ser entendido. Parece-nos que seria mais apropriado manter o uso do tratamento de exceção restrito aos "fios" de programação sequencial, e utilizar técnicas como troca de mensagens para interação entre processos concorrentes. Desta forma os pontos onde tal interação pode ocorrer ficariam claros tanto para o programador como para qualquer pessoa que tentasse entender o texto de um programa.

IV.8. Eiffel

A linguagem Eiffel permite que se introduzam assertivas lógicas em seus programas. Estas assertivas, basicamente compostas por expressões booleanas, podem ser verificadas em tempo de execução caso tal seja determinado no momento de compilação. Assertivas podem ser usadas para a verificação de pré e pós condições, ou ainda de uma outra condição qualquer. Caso haja violação de uma assertiva, considera-se que ocorreu uma exceção. Assim, exceções em Eiffel não são entidades declaradas, podendo mesmo existir sem receber nome algum, e não possuem parâmetros. Sua sinalização não é feita através de um comando explícito, mas

ocorre automaticamente quando da violação de alguma assertiva. Para levantar explicitamente uma exceção, é necessário definir uma rotina **raise** com pré-condição **false**.

O tratador de uma exceção é associado ao corpo de uma rotina, sob a forma de uma cláusula **rescue**. Dentro desta cláusula pode ser incluído o comando **retry**, que faz com que a rotina corrente volte a ser executada do início. Com isto permite-se que o programador tente restabelecer uma condição violada e ordenar uma nova tentativa de execução da rotina. Caso não seja incluído o comando **retry**, o final da execução da cláusula **rescue** fará com que a rotina corrente termine com falha. Desta forma, pretende-se impedir que o programador utilize o mecanismo de exceções para construir rotinas multi-saída. No entanto, o próprio autor da linguagem dá em [Meyer 88b, pag 155] um exemplo de rotina para calcular o inverso de um número onde se utiliza o comando **RETRY** acoplado a uma variável sinalizadora para retornar um valor especial no caso do número ser zero. Isto vem reforçar a dificuldade de distinguir os dois primeiros conceitos de exceção identificados no final da seção II.

Também é oferecida a opção de incluir uma cláusula **rescue** na definição de uma classe (a linguagem segue o paradigma de orientação a objetos), que é acionada caso ocorra uma exceção sobre uma instância da classe e a rotina sinalizadora não possua uma cláusula **rescue**. Esta facilidade nos remete às exceções de estrutura definidas por Levin, com a diferença de que em Eiffel não é a natureza da exceção que determina de que maneira ela será tratada, e sim o fato de existir ou não um tratador associado à rotina onde ocorre a exceção.

V. Escolha de um mecanismo

Acreditamos que a discussão desenvolvida nas seções anteriores indica que um mecanismo de tratamento de exceções pode simplificar bastante o trabalho do programador e tornar o resultado deste trabalho mais fácil de entender e modificar. Na seção IV estudamos os mecanismos existentes de forma crítica. Discutiremos agora uma proposta de mecanismo resultante deste estudo. A intenção é apenas delinear algumas idéias que nos parecem relevantes, e não apresentar uma descrição completa de mecanismo. Nossa proposta é fundamentalmente baseada na da linguagem CLU; Apesar de oferecer menos recursos do que outras propostas vistas, parece-nos que este mecanismo combina um poder suficiente para resolver a maior parte dos problemas levantados na literatura com uma simplicidade que torna seu uso confortável para o programador. No que segue defenderemos este ponto de vista e faremos algumas propostas de alterações. A discussão será organizada segundo os mesmos pontos pelos quais nos pautamos para discutir as propostas existentes.Após a discussão destes pontos, será apresentado um exemplo de aplicação do mecanismo.

V.1. Declaração de exceções

A utilidade de algum tipo de declaração de exceções independente dos cabeçalhos de procedimentos que as sinalizam foi discutida na descrição da proposta de MESA, na seção anterior. Como dito, a falta de uma tal declaração gera o problema de "colapso" de exceções com um mesmo nome, geradas por procedimentos diferentes, em uma só. Por outro lado, como também já observado anteriormente, a incorporação de um construtor de tipos exceções aumenta em muito a complexidade da linguagem. Uma solução que traria uma proteção a mais para o programador sem obrigá-lo a lidar com esta complexidade seria a seguinte. Cada módulo teria que declarar as exceções sinalizadas por seus procedimentos em uma seção especial. As exceções sinalizadas por procedimentos exportados (definidos na interface) teriam que ser exportadas também. A partir daí, ficaria simples para o compilador detectar

se em determinado bloco de programa o mesmo nome de exceção estivesse sendo importado de dois módulos diferentes e, neste caso, exigir a qualificação da exceção com o nome do módulo exportador. Com esta solução, põe-se fim ao problema de uma exceção sinalizada por um procedimento A ser "inadvertidamente" associada com um tratador pensado para outro procedimento B, desde que A e B pertençam a módulos diferentes. No caso deles pertencerem ao mesmo módulo, parece-nos que a probabilidade do programador realmente desejar tratar a mesma exceção da mesma forma, independentemente de qual dos dois a sinalizou, passa a ser suficientemente grande para justificar o risco do erro possível. A qualificação dos nomes das exceções com os nomes das unidades que as declaram encaixa-se particulamente bem na sintaxe de CLU, onde todos os procedimentos chamados são qualificados desta maneira.

V.2. Associação de uma exceção a um tratador

O enfoque dado por CLU à propagação automática de exceções parece-nos adequado. Por um lado, como já argumentado anteriormente, não faz sentido um procedimento receber a sinalização de uma exceção sobre a qual ele não tem o menor conhecimento. Por outro lado, a proibição total da propagação de exceções torna a linguagem um pouco rígida demais. Se a exceção sinalizada é algo bastante genérico (que pode ser interpretado como "tudo saiu errado") passa a fazer sentido que se procure em vários níveis um tratador capaz de lidar com o "desastre". A idéia passa a ser: quantas são as camadas que devem ser jogadas fora se um tal desastre acontecer?

Uma pequena modificação que faríamos ao mecanismo de CLU seria eliminar a diferença entre exceções locais, geradas com o comando **exit**, e exceções sinalizadas por rotinas. Parece-nos que os conceitos são semelhantes demais para que sejam usados dois comandos distintos. Os autores sustentam que comandos distintos devem ser usados para refletir o fato de que o programador tem intenções diferentes nos dois casos e que restrições mais rígidas devem ser colocadas no caso de exceções locais (por exemplo deve obrigatoriamente existir um tratador). No entanto, o simples fato

da exceção não estar declarada no cabeçalho do procedimento já reflete esta diferença e permite impor as restrições necessárias. Seria mais coerente, portanto, que se utilizasse o comando **signal** nos dois casos.

V.3. Fluxo de controle

Em relação à discussão retomada X terminação, a argumentação em [Liskov 79], onde são apresentadas soluções por terminação a problemas classicamente tratados por retomada, parece-nos indicar que a inclusão de retomada não acrescenta poder suficiente a um mecanismo de tratamento de exceções para justificar a complexidade e o perigo de falta de abstração que gera. Se um módulo M chama um procedimento D de um módulo N e D sinaliza uma exceção que pode ser tratada por M com retomada, não é mais possível dizer que a transformação de estado decorrente da execução de D depende apenas do estado interno de N e dos parâmetros passados por D, pois haverá interferência do estado interno de M (conforme observação em [Cristian 79]). A função do tratador no caso da retomada é restabelecer alguma assertiva que tenha sido violada (tornar possível a execução de uma operação diagnosticada como impossível). Se esta é uma assertiva de entrada, faria mais sentido que o módulo invocador realizasse uma nova chamada com parâmetros válidos. Se não é uma assertiva de entrada, então para restabelecer sua validade M deverá conhecer a implementação de D, em desacordo com os princípios de modularidade.

Ainda contra a retomada, deve-se ter em mente que, após a execução de um tratador, a rotina sinalizadora terá sempre que testar novamente a condição em questão, uma vez que o tratador pode não ter conseguido eliminá-la. No caso do resultado do teste ser positivo, uma exceção mais forte terá que ser sinalizada, ou um loop inserido para garantir um estado final não excepcional. Assim, a própria programação da rotina sinalizadora torna-se mais complexa com esta possibilidade.

Dada a opção pelo modelo de terminação, resta a questão do fluxo de controle após execução do tratador. O mecanismo de CLU tem em comum com muitos outros a ligação entre a posição física do

tratador (cláusula **when**) e o fluxo de controle após a execução do mesmo. A crítica de Yemini a esta ligação parece procedente a primeira vista; no entanto, uma análise dos exemplos dados na literatura mostra que, em geral, estas duas escolhas estão realmente interligadas. A desvinculação das duas traz o desconforto de uma solução como a sugerida por Yemini, que introduz uma "falsa exceção" para sair de um bloco. Parece-nos que o ônus de repetir uma cláusula **when** em vários pontos de um bloco nos casos em que não de deseje a terminação do mesmo se justifica por tratar-se do caso menos freqüente. Além disto, na proposta de Yemini esta independência entre fluxo de controle e a posição do tratador faz sentido porque no modelo de substituição proposto toda chamada de procedimento retorna um valor final, e portanto o fluxo pode prosseguir como se a chamada houvesse sido executada normalmente. No modelo de CLU, se uma exceção ocorre em uma chamada de procedimento dentro de uma expressão, como no comando

```
a := P(x) + 4
```

não haverá valor retornado por P(x), e portanto o fluxo de controle terá que ser alterado obrigatoriamente. Este sendo o caso, a solução de associá-lo à posição do tratador parece-nos facilitar o entendimento do mecanismo.

V.4. Um exemplo

A figura 1 mostra trechos de um programa onde é usado o mecanismo de exceções proposto acima. A linguagem utilizada é CLU, com as seguintes modificações: exceções são declaradas e exportadas pelos **clusters** que as sinalizam e o comando **signal** é utilizado ao invés de **exit**.

```
fila = cluster is create, insert, next, empty
       signals nomore

  rep = array[int]
  ...
  create = proc() returns (cvt)
  ...
  end create

  insert = proc(f: cvt, e: int)
  ...
```

```
    end insert

    next = proc(f: cvt) returns (int)
                        signals nomore
      if rep$low(p) < rep$high(p) then
        signal nomore
      ...
    end next

    empty = proc(f: cvt) returns (bool)
    ...
    end empty

  end fila

        = cluster is create, pop, push, empty
          signals nomore

    rep = array[int]

    create = proc() returns (cvt)
    ...
    end create

    push = proc(p: cvt, e: int)
    ...
    end push

    pop = proc(p: cvt) returns (int)
                       signals nomore
      if rep$low(p) = 0 then
        signal nomore
      ...
    end pop

    empty = proc(p: cvt) returns (bool)
    ...
    end empty

  end pilha

1  exemplo = proc() returns (tresult)
2                  signals caso1
3
4    i: int
5    f: fila := fila$create()
6    p: pilha := pilha$create()
7    ...
8    while true do
9      if "condição" then
         ...
15       i:= pilha$pop(p)
16         except when pilha$nomore: signal faltouinfo end
         ...
21       i:= fila$next(p)
22         except when fila$nomore: i:= pilha$pop(p) end
```

```
23      end
        ...
30      i := fila$next(p)
        ...
33      if "condicao" then signal caso1
        ...
35      i := pilha$pop(p)
        ...
40    end except
41      when fila$nomore:
          ...
45      when pilha$nomore:
          ...
50      when faltouinfo:
          ...
55  end exemplo.
```

Observe que não foram incluídas seções de declaração de exceções nos clusters. Isto porque estes não possuem uma cláusula de exportação em separado, e portanto a declaração de exceções no cabeçalho de um cluster equivale a uma seção deste tipo.

Se a condição **nomore** for sinalizada na chamada a **pilha$pop** na linha 15, o tratador na linha 16 será ativado. Este, por sua vez, sinaliza a condição **faltouinfo**, fazendo com que o controle de execção passe para a linha 50. Caso a condição **nomore** seja sinalizada pela chamada a **fila$next** na linha 21, o tratador da linha 17 será ativado. Por outro lado, as chamadas a **fila$next** e **pilha$pop** nas linhas 30e 35 estão no escopo dos tratadores de exceção associados ao bloco de repetição inteiro. Caso uma destas chamadas sinalize a condição, o controle irá para a linha 41 ou 45. Observe que seria ilegal escrever "**when** nomore" na linha 41, sem qualificação do cluster sinalizador.

Se **caso1** for sinalizada, na linha 33, esta condição será reconhecida como pertencente à interface da rotina, fazendo com que a rotina **exemplo** sinalize caso1. Observe que, se houvesse um tratador para **caso1** associado ao bloco de repetição, este seria ativado. No entanto, uma boa disciplina de programação indicaria o uso de nomes diferentes para exceções "locais" e "globais", isto é, exceções cuja sinalização implicam simplesmente na terminação de um comando na rotina corrente e aquelas cuja sinalização implicam na terminação da própria rotina corrente.

VI. Considerações Finais

Apesar do termo exceção já estar sendo usado há mais de 20 anos, não existe um consenso sobre seu significado. Na seção II fizemos um levantamento do sentido dado a este termo pelos vários autores que o vêm usando, e finalizamos aquela seção com uma classificação de "situações excepcionais". Ao longo do resto do trabalho, principalmente na seção IV e V, discutimos as desvantagens do uso do conceito de exceção como "entrelaçamentode níveis". Optamos então pelo não oferecimento de uma facilidade com este fim. Quanto aos dois primeiros usos observados do termo, o de um estado final inadmissível para um procedimento e o de um entre vários estados finais possíveis, já foi observado quão frágil é a fronteira entre os eles. Parece-nos que o primeiro caso decididamente merece um mecanismo especial, enquanto o segundo pode ser tratado bem por outras construções; no entanto, é quase impossível oferecer uma facilidade que possa ser usada no primeiro caso e não no segundo, e a decisão de usá-la ou não parece-nos ser mais uma questão de escolha de uma disciplina de programação. Apontamos então o mecanismo de tratamento de exceções de CLU como um bom modelo, por combinar poder suficiente com a simplicidade necessária para ser realmente utilizado de maneira proveitosa por um programador.

Um trabalho que poderia ser levado adiante seria uma investigação mais profunda da opção contra a facilidade de retomada. Para afirmar que esta facilidade não é crucial, baseamo-nos em alguns exemplos dados na literatura e em nosso bom senso. No entanto, poderia ser interessante realizar-se um estudo extensivo de casos e demonstrar-se a suficiência do mecanismo de terminação.

Bibliografia

[Black 82] Black, Andrew.
Exception Handling: the Case Against. University of Oxford, 1982. Tese de Doutorado.
[Cheriton 86] Cheriton, David.
Making Exceptions Simplify the Rule (and Justify their handling). *Proc. IFIP Congress 86*, Dublin, Ireland, 1986.
[Cristian 79] Cristian, Flaviu.
Le Traitement des Exceptions dans les Programmes Modulaires. Grenoble,Universidade de Grenoble, 1979. Tese de Doutorado.
[Cristian 82] Cristian, Flaviu.
Exception Handling and Software Fault Tolerance. *IEEE Trans. on Computers*. *31*(6) 531-540, jun 1982.
[Dijkstra 76] Dijkstra, E.W.
An Essay on the Notion: "The Scope of Variables". In *A Discipline of Programming*. New Jersey, Prentice-Hall, Englewood Cliffs, 1976.
[Eggert 81] Eggert, Paul R.
Detecting Software Errors Before Execution. Los Angeles,UCLA, 1981. Tese de Doutorado (disponível como relatório técnico UCLA CSD-810402).
[Goodenough 75] Goodenough, J.B.
Exception handling: issues and a proposed notation. *Comm. ACM 18*(2), dec 1975.
[Ichbiach 79] Ichbiach, J. et alii.
Rationale for the design of the Ada programming language", *ACM SIGPLAN Notices*. *14*(6), jun 1979.
[Knuth 74] Knuth, Donald..
Structured programming with go to statements. *Computing Surveys*. *6*(4) 261-301, dec 74.
[Levin 77] Levin, Roy.
Program structures for exceptional condition handling. Pittsburgh, PA, Carnegie-Mellon University, jun 1977. Tese de Doutorado.
[Liskov 79] Liskov, Barbara. Snyder, Alan.
Exception Handling in CLU. *IEEE Trans. on Software Engineering*. *SE-5*(6) 546-558, nov 1979.
[Meyer 88a] Meyer, Bertrand.
Eiffel: a language and environmente for software engineering. *The Journal of Systems and Software*, *8*(3), 199-246, jun 1988.
[Meyer 88b] Meyer, Bertrand.
Object-Oriented Sofware Construction. Prentice-Hall, Englewood Cliffs, N.J., 1988.
[Mitchell 79] Mitchell, J.G.; Maybury, W.; Sweet, R.
MESA Language Manual. XEROX Palo Alto Research Center, mar 1979.
[Noble 68] Noble, J. M.
The Control of Exceptional Conditions in PL/1. *Proc. IFIP Congress 68* C78-C83, ago 1968.
[Randell 75] Randell, Brian.
System Structure for Software Fault Tolerance, *IEEE Trans.Sofware Engineering*. *SE-1*(2) 220-232, jun 1975.
[Strom 86] Strom, Robert. Yemini, Shaula.
Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. on Software Engineering*. *SE-12*(1) 157-171, jan 1986.
[Wulf 76] Wulf, W.A.

Abstraction and Verification in Alphard. In S.A. Schuman(ed).*New Directions in Programming Languages - 1975*. IRIA, 1976.

[Yemini 80] Yemini, Shaula.
The replacement model for modular verifiable exception handling. Los Angeles, UCLA, 1980.Tese de Doutorado.

[Yemini 85] Yemini, Shaula. Berry, Daniel.
A Modular Verifiable Exception Handling Mechanism. *ACM Trans. on Computer Languages and Systems*. 7(2) 214-243, apr 1985.

[Zahn 74] Zahn, Charles.
A control statement for natural top-down structured programming. In B. Robinet (ed). *Programming Symposium*. New York, Springer-Verlag, 1974.Lecture Notes in Computer Science 19.