Uma Experiência na Implementação de um Sistema de Tipos Fortes e Polimórficos

Walace de Almeida Rodrigues
Roberto da Silva Bigonha
Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais
e-mail: walace@dcc.ufmg.br
e-mail: bigonha@dcc.ufmg.br

Resumo

Uma das principais vantagens das linguagens fortemente tipadas é que em programas aceitos pelo compilador não poderão ocorrer erros de tipo durante a fase de execução. A desvantagem é que esta característica limita o domínio de aplicação das funções definidas na linguagem. Por outro lado, linguagens de programação modernas defendem a possibilidade de se ter polimorfismo, o qual é um recurso poderoso que permite ao programador definir funções que trabalham com argumentos de diversos tipos.

Este artigo relata a experiência adquirida no projeto e implementação de um *type-checker* para a linguagem SDL utilizada para escrita de definições denotacionais de semântica. O sistema de tipos desta linguagem é de especial interesse uma vez que permite o polimorfismo associado a uma disciplina de tipos fortes.

[keywords]: semântica denotacional, SDL, polimorfismo, *type-checking*, equivalência de tipos.

Abstract

One of the benefit of strongly typed languages is that once programs has been accepted by the compiler, no errors can occur during execution. A disadvantage of this type of languages is the restriction it imposes on the domain of application of the user's defined functions. On the other hand, modern programming languages implement polymorphism, which is a powerful mechanism that allows functions to operate on several type of arguments.

This paper reports the experience in the design and implementation of a type checker for the denotational semantics language SDL, which combines a strong type discipline with polymorphism.

1 Introdução

A linguagem SDL [4] é utilizada para escrita de especificações formais de semântica de linguagens de programação e faz uso do método denotacional [14, 10, 19, 7, 18, 8] como forma de especificação formal.

Um compilador para a linguagem SDL foi implementado [1] no Departamento de Ciência da Computação da UFMG como parte integrante de um ambiente para a definição de semântica legível de linguagens de programação [4, 2, 9, 1]. Neste ambiente, chamado **xlds**, uma definição semântica oferece como saída, entre outros sub-produtos, um compilador para a linguagem descrita. O código objeto gerado pelo compilador SDL é um programa escrito na linguagem LAMB, que nada mais é que uma representação formal para *lambda-calculus*. A geração automática dos compiladores por parte do sistema **xlds** facilita os testes de consistência e torna as definições semânticas mais confiáveis.

Ainda com o objetivo de tornar mais consistente as definições, além de também permitir a geração de código mais eficiente, a linguagem SDL foi projetada de forma a utilizar um sistema de tipos que é um compromisso entre a filosofia de linguagens fortemente tipadas [6], como ALGOL 68, e de linguagens onde nenhuma disciplina de tipos é imposta, como LISP. O objetivo foi ter as bem conhecidas vantagens das linguagens fortemente tipadas retendo, ao mesmo tempo, a flexibilidade das definições de funções que trabalham com uma larga variedade de tipos. C.Strachey usou o termo *polimorfismo* para descrever funções com esta característica.

O polimorfismo utilizado na linguagem SDL foi baseado na proposta de Milner [11] que foi extensamente modificada em [4] para poder ser aplicada ao caso SDL. Também deve ser lembrado que apesar de seu polimorfismo a linguagem SDL é fortemente tipada e usa uma forma de equivalência estrutural [1] para decidir se dois tipos são ou não equivalentes num dado contexto.

2 Domínios

Na linguagem SDL todos os tipos são representados por domínios [14]. Domínios são entidades matemáticas representando conjuntos parcialmente ordenados com um elemento mínimo chamado **bottom** que é usualmente representado por $\perp$. Nesta apresentação não entraremos em considerações matemáticas acerca da teoria dos domínios, tais informações poderão ser encontradas em [14, 15, 17, 5, 8]. O valor $\perp$ não tem representação direta em SDL, ele serve para modelar a semântica de não-terminação (loop). Por exemplo, $\perp$ pode ser tomado como o menor ponto fixo [17] de uma equação recursiva do tipo $f(x) = f(x+1)$. Além deste valor, todo domínio SDL conta com outro elemento especial chamado **undefined** que é representado por ? . Este símbolo é usado para indicar o valor de uma expressão SDL semanticamente sem sentido.

2.1 Domínios primitivos

Os domínios primitivos em SDL são domínios básicos já implementados na linguagem e disponíveis para o usuário. São estes:

N = domínio básico dos inteiros não-negativos
Q = domínio básico das **quotations**, ou **strings**
T = domínio básico dos valores lógicos
E = domínio básico das expressões LAMB
P = domínio básico das árvores de **parser**

2.2 Constantes

Em SDL as constantes literais podem ser:

- Números decimais representando membros do domínio N. Por exemplo: 0, 1, 2, ...
- Valores lógicos membros do domínio T. Os valores TT e FF denotam verdadeiro e falso respectivamente.
- **Quotations** representando membros do domínio Q. Por exemplo: "UM EXEMPLO".
- O valor **undefined** que é representado por ? e é membro de todo domínio SDL.

2.3 Domínios definidos pelo usuário

SDL oferece uma variedade de operadores que permitem ao usuário criar novos domínios e modelar propriedades sintáticas ou semânticas de diferentes linguagens. Uma expressão de domínio pode ser um nome de domínio, uma **quotation** denotando um domínio cujo único elemento é a constante, uma combinação de expressões de domínios e operadores de domínios. Se $d_1, \ldots, d_n$ são expressões de domínio então novas expressões de domínio podem ser construídas da forma mostrada aqui:

- $d_1 | \ldots | d_n$ denota a união dos domínios $d_1, \ldots, d_n$. SDL usa a união (|) em oposição à soma-separada (+) de Scott [14]. Na opinião de Mosses [12] tal método tem a vantagem de libertar o usuário da obrigação de tratar com projeção e injeção de domínios e seus elementos. Note contudo que há uma exigência para que os domínios operandos da união possam ser diferenciados pelo uso do operador IS, definido em SDL e encarregado de fazer o *pattern-matching* de estruturas de domínios.
- $(d_1, \ldots, d_n)$ denota uma tupla cujo i-ésimo componente é o domínio d_i para $1 \leq i \leq n$. Esta é a notação em SDL para o Produto Cartesiano de domínios.
- $d+$ denota uma tupla finita e não-vazia cujos componentes estão em d.
- $d*$ denota uma tupla finita e possivelmente vazia cujos componentes estão em d.
- $d_1 \rightarrow d_2$ denota uma função contínua que mapeia elementos de d_1 para d_2.
- $[d_1 \ldots d_n]$ denota o domínio dos nodos das árvores de **parser** com mesmo **label** que o definido pela **quotation** QUOTE $(d_1, \ldots, d_n)$.
  Um nodo da árvore de **parser** consiste em um **label**, representado por uma **quotation**, e uma tupla que contém os ramos que partem do nodo. O i-ésimo ramo partindo do nodo $[d_1 \ldots d_n]$ é um membro do domínio d_i para $1 \leq i \leq n$.

O operador [...] requer que os domínios $d_1 \ldots d_n$ sejam constantes literais ou identificadores de domínios, possivelmente acompanhados dos operadores '$*$' ou '$+$'.

- (d) denota o próprio domínio d.

Em SDL, a sintaxe para introduzir um novo domínio A é:

```
x : A = d
```

onde `x` é uma variável em `A`, `A` o nome do domínio, e `d` a expressão que o descreve. Cabe ainda ressaltar que qualquer dos termos desta construção é opcional, desta forma a construção `y := d` apenas define a variável `y` como pertencendo ao domínio definido pela expressão `d`, sem atribuir um nome a este domínio. De forma análoga, a construção `A = d` introduz o novo domínio `A`, mas sem definir nenhuma variável.

2.4 Domínios Variáveis

Domínios variáveis são usados para indicar tipos de funções polimórficas e operadores. Estes domínios são representados pelo caracter '@' seguido de um número inteiro. Como exemplo tomemos o caso da função identidade

```
id = λ x . x
```

cujo tipo pode ser especificado como

```
id := @1 → @1
```

denotando uma função polimórfica que pode receber argumentos de qualquer domínio. Na seção que trata do polimorfismo, os domínios variáveis serão discutidos em maior detalhe.

3 Polimorfismo

A palavra *tipo* é usada aqui para denotar um domínio SDL. Tipos variáveis são representadas pelo caracter '@' seguido de um número não-negativo. Domínios cuja denotação não contém tipos variáveis são chamados **monodomínios** e em caso contrário são chamados **polidomínios**.

Uma clara distinção deve ser feita entre domínios e domínios variáveis: um domínio atua como uma constante, isto é, representa sempre a mesma denotação de domínio; uma variável de domínio, por outro lado, pode representar diferentes denotações de domínio. Por exemplo, através das expressões

$A = N \rightarrow N;$
$f := @1 \rightarrow @1;$

fica entendido que o domínio A está sempre ligado a $N \rightarrow N$ enquanto que o valor associado com @1 depende do tipo do argumento que será passado para f em uma aplicação da função. Desta forma o valor de @1 pode mudar se f for usada mais de uma vez.

A seguir apresentamos as regras necessárias à implementação da disciplina de tipos da linguagem SDL.

3.1 Instanciação de variáveis de tipo

Em uma expressão de domínio toda instância de uma mesma variável é associada com um mesmo tipo. Por exemplo, na expressão

$$f := @1 \rightarrow @2 \rightarrow (@1, @2);$$

o identificador f deve ser interpretado como uma função que recebe dois argumentos de tipos quaisquer e retorna uma tupla de dois componentes do mesmo tipo dos argumentos.

Regra 1 *Toda instância de uma mesma variável em uma expressão de domínio designa o mesmo tipo.*

Variáveis de tipo que ocorrem em diferentes expressões de domínio são não-relacionadas ainda quando tomam o mesmo tipo. Por exemplo, nas expressões

$$f1 := @1 \rightarrow @1;$$
$$f2 := @1 \rightarrow @1;$$

as ocorrências de @1 nos tipos de $f1$ e $f2$ não têm qualquer relação entre si. Os identificadores $f1$ e $f2$ denotam funções cujos domínios retornados são iguais aos domínios recebidos como argumento, o fato de @1 ser ligado a N em $f1(x : N)$, por exemplo, não força o tipo de $f2$ a ser $N \rightarrow N$. É evidente que a intenção ao definir os tipos de $f1$ e $f2$ não era torná-los dependentes e o fato de ambos os tipos usarem o mesmo nome para uma variável de domínio é somente coincidência. Assim a linguagem SDL considera a variável @1 que aparece no tipo de $f1$ como uma variável diferente daquela que aparece no tipo de $f2$.

A idéia básica por trás das variáveis de domínio é permitir a definição de tipos que tenham um certo grau de liberdade e a primeira regra é muito importante porque é ela que determina a estratégia geral a seguir na determinação do tipo das aplicações funcionais.

Por outro lado, considere agora o seguinte exemplo:

$$A = @1;$$
$$B = @1;$$
$$h := A \rightarrow B;$$

Neste caso a interpretação mais natural para o tipo de h é $@1 \rightarrow @1$ mas isto é exatamente o oposto de considerarmos as duas instâncias de @1 independentes. Não fica claro qual é a melhor interpretação e então decidiu-se que casos como o do exemplo acima seriam proibidos porque complicam muito o mecanismo da verificação de tipos. Portanto as especificações de tipo requerem equações que definem monodomínios, isto é, um identificador de domínio não pode ser ligado a expressões de tipo que contém variáveis de domínio. Esta restrição não diminue o poder de expressão da linguagem e também simplifica o mecanismo que determina o tipo de funções polimórficas.

3.2 Primeiro exemplo

Considere agora o exemplo das especificações dos domínios das funções para manipular listas arbitrárias em SDL:

```
m      := @1*;
hd     := @1* → @1;
tl     := @1* → @1*;
empty  := @1* → T;
f      := @1 → @2;
map    := (@1 — @2, @1*) → @2*;
```

Aqui o domínio `m` denota uma lista de elementos de um tipo arbitrário. As funções `hd` e `tl` retornam respectivamente o **header** e o **tail** da lista e a função `empty` retorna um valor lógico que indica se a lista é vazia ou não. A função `f` mapeia valores do domínio @1 para valores do domínio @2. A função `map` recebe como argumentos uma função do mesmo tipo de `f` e uma lista com elementos do mesmo tipo dos argumentos de `f`, o valor retornado por `map` é uma lista com elementos do mesmo tipo dos valores retornados por `f`.

Note como as expressões polimórficas relacionam os domínios, este relacionamento constitui a base da disciplina de tipos da linguagem SDL.

3.3 O processo de substituição

Para determinar o tipo de uma expressão o **type checker** deve primeiro associar um domínio a toda ocorrência de variável e então avaliar o tipo da expressão resultante. Desde que variáveis de domínio em diferentes expressões são em princípio não-relacionadas então elas devem ser renomeadas de acordo para prevenir contra a colisão de tipos. Este processo é chamado de **substituição**, e a expressão de tipo obtida após o processo é chamada **instância de substituição** da expressão original. Por exemplo, a expressão @10 → @11 é uma possível instância de substituição do domínio `f`[1].

3.4 O processo de explicação

Considere agora os domínios mostrados no exemplo 3.2 e suponha que estes tenham as instâncias de substituição

```
m      : @10*;
f      : @10 → @11;
map    : (@12 → @13, @12*) → @13*
```

O tipo da expressão `map(f,m)` é de fato @13*, mas é necessário também manter a informação de que a variável @13 é simplesmente um sinônimo para @11, isto é, neste contexto as variáveis @13 e @11 são a mesma variável. De modo análogo chegamos às variáveis sinônimas @12 e @10.

[1] Ver exemplo 3.2.

Agora basta trocar a variável @13 por seu sinônimo e obtemos o tipo final da expressão `map(f,m)` que é @11*. O processo de trocar as variáveis de tipo por seus sinônimos ou pelos domínios que denotam é chamado de **explicação**.

3.5 Tipos de Parâmetros Formais

Assuma agora que a função `map` mostrada no exemplo 3.2 é definida da forma

DEF `map(f,m) = empty(m) → ( ), f(hd(m)) PRE map(f,tl(m));`

Note que o lado esquerdo desta equação também fornece o tipo dos parâmetros da função `map` e que estes devem ser compatíveis com a especificação que aparece no exemplo. O argumento passado para a função `map` é uma tupla cujos componentes são os tipos de `f` e `m` e se suas instâncias de substituição são respectivamente @10 → @11 e @12* então o domínio de `(f,m)` é (@10 → @11, @12*).

Aqui surgiu um problema. Este domínio não casa com o tipo que aparece na especificação e embora ambos tenham a mesma estrutura o domínio acima não mostra qualquer relação entre os tipos do primeiro e segundo componentes da tupla. Na verdade o problema surgiu após a substituição dos domínios de `f` e `m` uma vez que este processo tornou-os independentes entre si. A solução é criar uma exceção na lei de escopo para variáveis de tipo quando tratando o lado esquerdo de equações que definem funções como a do exemplo 3.2.

Regra 2 *O tipo dos parâmetros formais definidos por LET ou DEF devem ser substituídos juntos como se seus tipos fizessem parte de uma mesma expressão de domínio.*

A variável @1 que aparece nos tipos de `f` e `m` devem ser consideradas como sendo a mesma variável de domínio e assim, com o propósito de determinar o tipo do lado esquerdo de uma equação, os tipos de `f` e `m` devem ser substituídos para @10 → @11 e @10* respectivamente. Desta forma o tipo de (f, m) torna-se (@10 → @11, @10*) e este resultado equivale ao tipo (@1 → @2, @1*) porque ambos têm a mesma estrutura e mostram a mesma relação entre os tipos.

Consideremos agora o lado direito de uma equação. No caso da função `map` que foi definida na seção 3.5 aparecem identificadores livres (como `hd`, `tl`, `empty`), parâmetros formais (como `f`, `m`) e uma ocorrência recursiva (a chamada da função `map`). Desde que o objetivo de um **type checker** é determinar o domínio das expressões por meio da relação entre os tipos do corpo e dos argumentos das funções então um mesmo domínio deve ser associado às ocorrências de um mesmo parâmetro formal. O sistema de tipos requer que variáveis de domínio envolvidas no tipo dos parâmetros também participem no tipo do corpo da função correspondente.

Regra 3 *Ocorrências de um parâmetro formal devem ter exatamente o mesmo tipo, isto é, o tipo de um parâmetro formal é substituído somente uma vez e toda ocorrência de um mesmo parâmetro formal, incluindo aquelas que aparecem no lado esquerdo da equação, é associada com a mesma instância de substituição.*

Desta forma concluímos que os tipos associados a todas as ocorrências de `f` e `m` que aparecem na função `map` devem ser iguais.

3.6 Tipos em Expressões Condicionais

Parece razoável exigir que o tipo retornado por uma expressão condicional não dependa do resultado da expressão pois, em caso contrário, a verificação dos tipos não poderia ser executada em tempo de compilação.

Regra 4 *Uma expressão condicional retorna o mesmo tipo da expressão que seria executada em consequência e as duas expressões que poderiam ser executadas, caso da condição ser FF ou TT, devem ser do mesmo tipo.*

Como exemplo considere a definição da função `map` [2] e as instâncias de substituição

```
m     := @10*;
hd    := @14* → @14;
tl    := @15* → @15*;
empty := @16* → T;
f     := @10 → @11;
map   := (@12 → @13, @12*) → @13*;
```

Com as informações anteriores concluímos que se a condição resultasse falsa chegariamos a expressão `f(hd(m)) PRE map(f,tl(m))` do tipo @11* [3]. Por outro lado uma condição verdadeira nos levaria a tupla vazia ( ) e esta, de acordo com a regra acima. deve ser associada com o mesmo tipo da expressão anterior, ou seja, o tipo @11*. Daí concluímos que o lado direito da equação que define a função `map` tem o tipo @11*.

3.7 Tipos em Definições Recursivas

Pelo que foi exposto na regra 3 não é difícil deduzir que o uso da recursividade na definição de funções polimórficas também deve ser governada por uma regra similar.

Regra 5 *Diferentes ocorrências de um identificador sendo recursivamente definido devem ter o mesmo tipo.*

Assim toda ocorrência de `map` que aparece na definição da função [4] deve ser associada à mesma instância de substituição.

Considerando que o tipo da tupla `(f,m)` é (@10 → @11, @10*) e que o tipo da expressão `f(hd(m)) PRE map(f,tl(m))` é @11* [5] podemos concluir que o type checker vai associar o tipo (@10 → @11, @10*) → @11* à função `map`.

Tudo que foi exposto acima está de acordo com a especificação e se considerarmos o lado esquerdo de uma equação como sendo uma aplicação da função então concluímos que `map` aceita os argumentos `(f,m)` e que `map(f,m)` retorna o tipo @11*. Este resultado está de acordo com a seguinte regra:

Regra 6 *Ambos lados de uma definição recursiva devem ter o mesmo tipo.*

[2] Ver seção 3.5.
[3] Não é difícil chegar a esta conclusão partindo das regras já enumeradas.
[4] Ver seção 3.5.
[5] Ver seção anterior.

3.8 Particularização de Tipos

Para o caso de definições não-recursivas a regra 6 é muito restritiva e parece conveniente aceitar que expressões do tipo

LET f = g IN .....;

sejam consideradas bem-tipadas se f e g forem do tipo @1 → @1 e $N \rightarrow N$ respectivamente. Do ponto de vista da verificação de tipos a definição acima pode ser considerada um meio de restringir o domínio de f para uso local, isto é, a função f seria considerada do tipo $N \rightarrow N$ no escopo particular da definição.

Regra 7 *O tipo do lado direito de uma definição não-recursiva deve ser igual ao tipo do lado esquerdo correspondente ou então uma forma particular dele.*

Um tipo X é considerado uma forma particular do tipo Y se X e Y têm a mesma estrutura mas Y contém mais variáveis de domínio que X. Por exemplo, o domínio $N \rightarrow N$ é uma forma particular do domínio @1 → @1. Para construirmos um forma particular de um tipo polimórfico Y substituímos todas as ocorrências de uma variável de tipo por um mesmo domínio. Deste modo o domínio $N \rightarrow N$ pode ser considerado uma forma particular do domínio @1 → @1 mas o mesmo não é verdade para $N \rightarrow T$.

3.9 Segundo exemplo

O exemplo abaixo vai ser útil para ilustrar o que acontece quando LET ou DEF são usados localmente, isto é, quando aparecem em uma abstração λ (*lambda-abstractions*). Considere a expressão e os tipos

LET tagpair = λ a . LET tag = pair(a) IN apply(tag,tag);

a := @1;
tag := @1 → (@2, @1);
pair := @1 → @2 → (@1, @2);
tagpair := @1 → (@2, @3) → ((@1, @2), (@1, @3));
apply := (@1 → @2, @3 → @4) → (@1, @3) → (@2, @4);

O modo como o type checker vai tratar o caso acima vai ficar mais claro após explicarmos a próximas regras a seguir.

3.10 Tipos em Abstrações

Assim como acontece com os parâmetros formais que aparecem nas definições funcionais – veja o caso dos parâmetros f e m que aparecem na definição da função map [6] – uma variável livre na abstração λ estabelece uma relação entre o tipo dos argumentos e o tipo que é retornado. Isto no conduz a seguinte regra:

[6] Ver seção 3.5.

Regra 8 *Ocorrências de uma variável que aparece como parâmetro de abstrações LAMB devem ser associadas com um mesmo tipo.*

3.11 Variáveis Genéricas

Vamos voltar agora ao segundo exemplo. Supondo que `pair` e `a` tenham como instâncias de substituição os domínios $@10 \rightarrow @12 \rightarrow (@10, @12)$ e $@13$ concluímos que a expressão `pair(a)` é do tipo $@12 \rightarrow (@13, @12)$. Agora, se a instância de substituição de `tag` for o domínio $@14 \rightarrow (@15, @14)$ então, pela regra 7, as variáveis $@14$ e $@12$ são sinônimas, assim como as variáveis $@15$ e $@13$, e concluímos que o tipo local de `tag` é $@12 \rightarrow (@13, @12)$. Desta forma chegamos a conclusão que a definição `tag = pair(a)` é bem tipada.

Suponha agora que o tipo da única ocorrência de `apply` é a instância de substituição $(@16 \rightarrow @17, @18 \rightarrow @19) \rightarrow (@16, @18) \rightarrow (@17, @19)$. Se cada ocorrência de `tag` for associada com uma instância de substituição diferente, digamos $@20 \rightarrow (@21, @20)$ e $@22 \rightarrow (@23, @22)$, então o tipo da expressão `apply(tag,tag)` que aparece na cláusula IN de definição da função será $(@16, @18) \rightarrow (@17, @19)$. Logo, teremos as variáveis sinônimas $@16$ e $@20$, $@17$ e $(@21, @20)$, $@18$ e $@22$, $@19$ e $(@23, @22)$ e depois da explicação o tipo da expressão `apply(tag,tag)` será $(@20, @22) \rightarrow ((@21, @20), (@23, @22))$.

Com o raciocínio acima concluímos que o tipo da expressão λ completa, e de `tagpair`, é $@13 \rightarrow (@20, @22) \rightarrow ((@21, @20), (@23, @22))$. No entanto, ao compararmos este tipo como a definição que aparece no exemplo, fica claro que alguma coisa de errado ocorreu. O segundo tipo tem um grau de polimorfismo maior que o primeiro e, embora os dois tenham a mesma estrutura, a variável $@13$ que aparece no primeiro domínio obedece a restrições que não existem no segundo.

O erro surgiu quando `tag` apresentou duas instâncias de substituição, uma para cada das duas ocorrências em `apply(tag,tag)`. Neste ponto se perdeu a informação de que o tipo de `tag` depende da variável `a`. Para resolver este problema Milner [11] sugeriu que o mecanismo de substituição fosse governado pela nova regra:

Regra 9 *No caso de substituição de variáveis que são argumentos de expressões LET ou DEF, somente aquelas que não aparecem em abstrações λ ou são parâmetros formais podem ser substituídas.*

Variáveis que podem ser substituídas são chamadas **variáveis genéricas** e consequentemente a substituição deve ser redefinida como sendo o renomeamento sistemático de todas variáveis genéricas envolvidas em uma expressão de tipo. Desta forma, de acordo com a regra 9, a primeira e a segunda ocorrências de `tag` em `apply(tag,tag)` devem ser associadas aos tipos $@20 \rightarrow (@13, @20)$ e $@21 \rightarrow (@13, @21)$ respectivamente. Isto nos leva a concluir que o tipo de `apply(tag,tag)` é $(@16, @18) \rightarrow (@17, @19)$ e que as variáveis $@16$ e $@20$, $@17$ e $(@13, @20)$, $@18$ e $@21$, $@19$ e $(@13, @21)$ são sinônimas.

Agora, depois de executado o processo de explicação, concluímos que o tipo da expressão `apply(tag,tag)` torna-se-á $(@20, @21) \rightarrow ((@13, @20), (@13, @21))$ e que o tipo de `tagpair` tornar-se-á $@13 \rightarrow (@20, @21) \rightarrow ((@13, @20), (@13, @21))$. Estes tipos são equivalentes aos tipos que aparecem na definição e concluímos que toda a expressão é bem tipada.

4 Equivalência de tipos

Como foi dito anteriormente o sistema de tipos utilizado na linguagem SDL faz uso de uma equivalência estrutural para decidir se dois domínios quaisquer são ou não equivalentes. De acordo com Welsh [21] a equivalência estrutural pode ser descrita da seguinte forma:

> "Duas variáveis podem ser consideradas do mesmo tipo se elas têm componentes do mesmo tipo estruturados do mesmo modo"

Evidentemente isto é muito geral e existem diversos algoritmos para testar a equivalência estrutural. O algoritmo que mostraremos aqui é uma variante dos métodos já existentes, maiores informações podem ser encontradas em [16, 20, 3, 21].

4.1 O caso SDL

A forma de equivalência estrutural que é utilizada na linguagem SDL é uma variação dos algoritmos de equivalência que são normalmente encontrados na literatura. Basicamente são usadas duas formas de equivalência que diferem entre si na forma como são tratados os domínios variáveis. Por exemplo, o domínio $@1 \rightarrow @1$ é somente **normalmente** equivalente a $@2 \rightarrow @2$ mas é **fortemente** equivalente a $@1 \rightarrow @1$.

Durante os testes com o compilador ficou claro que os algoritmos para tratamento de equivalência estrutural, tal como são normalmente definidos na literatura [16, 20, 3, 21], são muito restritivos. Estes algoritmos muitas vezes consideram como não-equivalentes tipos que intuitivamente o são. Por exemplo, os algoritmos citados acima consideram como não-equivalentes os tipos X e Y quando definidos da forma

$$A = N \rightarrow N;$$
$$X = (A, A);$$
$$Y = (N \rightarrow N, N \rightarrow N);$$

Para resolver este problema foi decidido extendermos os algoritmos de equivalência de forma que, quando checando identificadores de domínio com não-identificadores, levamos em consideração o domínio que o identificador denota. É evidente que isto pode levar a ocorrência de loops e também foi feito um tratamento neste sentido. Quando considerando a denotação de identificadores, se esta é também um identificador então colocamos uma marca neste para que o mesmo não seja 'visitado' novamente. Se o algoritmo nos conduz a uma denotação que é um identificador já 'visitado' então consideramos os tipos como não equivalentes. Assim, os algoritmos ficam definidos da forma descrita abaixo:

1. O domínio ? é equivalente somente a si mesmo. Por definição **undefined** tem o tipo que o contexto requer mas isto não implica que o domínio de valores indefinidos é equivalente a todo domínio.

2. Os domínios primitivos N, P, T e E são equivalentes somente a si mesmos.

3. O domínio Q dos literais e o domínio q de um literal particular são considerados equivalentes por razões práticas.

4. Dois domínios são equivalentes se e somente se suas denotações são equivalentes.

5. Os domínios *dom** e *dom+* são considerados equivalentes por razões práticas.

6. O domínio das tuplas vazias ( ) é equivalente somente a si mesmo. Por definição uma tupla vazia tem o tipo que o contexto requer mas isto não implica que o domínio das tuplas vazias seja equivalente a todo domínio de tuplas.

7. Duas uniões de domínios são equivalentes se e somente se seus respectivos operandos são equivalentes um a um na mesma ordem. Por exemplo, $A \mid B$ não é equivalente a $B \mid A$.

8. Um domínio da forma $A*$ ou $A+$ é equivalente ao domínio $(A_1, \ldots, A_n)$ se e somente se todos domínios A_i, para $1 \leq i \leq n$, são equivalentes a A.

9. O domínio A é equivalente ao domínio (A).

4.2 Tratamento de Polimorfismo

Além do algoritmo que testa a equivalência é também necessário mostrar os algoritmos para tratar e reconhecer polidomínios. Especificamente, os algoritmos que definem os mecanismos básicos para executar **explicações, substituições, unificações** e **particularizações de tipos.**

4.2.1 Explicação

O mecanismo de explicação consiste de mudar exaustivamente todas as variáveis de domínio, em uma dada expressão de tipo, pelos domínios associados.

Para tal deve ser utilizada uma tabela *ee* onde são armazenados todos os sinônimos para uma dada variável, inicialmente a tabela é definida no *environment* como sendo λ `dvar . [dvar]`. Por exemplo, a explicação de $(@1, @1)$ é (N, N) quando $ee(@1) = N$. A função que vai ser encarregada de retornar o domínio que a variável denota deve ser suficientemente inteligente para testar se as variáveis de domínio não são definidas recursivamente em *ee*, caso contrário a função entra em *loop*.

4.2.2 Substituição

O mecanismo de substituição, ou instanciação, já foi mostrado anteriormente, a função encarregada deste mecanismo vai receber um domínio *dom* como entrada e retornar um novo domínio *dom'* obtido pela substituição de todas as variáveis de domínio genéricas por variáveis de domínio únicas.

Uma tabela *rg* deverá ser utilizada para recordar todas as variáveis que foram substituídas assim como seus sinônimos [7]. Esta tabela é inicialmente definida no *environment* como sendo λ `dvar` . [TT], ou seja, inicialmente todas as variáveis são genéricas. A função que trata a substituição é que vai estar encarregada de manter esta tabela atualizada.

Esta tabela deverá ser também utilizada na definição de um *environment* que permite cumprir a regra 9 das regras que descrevem o sistema de tipos. Esta regra proibe a substituição de variáveis que aparecem no tipo de λ ou que são parâmetros formais.

4.2.3 Unificação

Durante a verificação de tipos em expressões SDL as variáveis de domínio podem ser associadas com muitas denotações diferentes. O propósito desta associação é particularizar o tipo de um dado domínio polimórfico ou determinar o tipo resultante de uma dada aplicação de funções polimórficas. A unificação de dois domínios explicados, digamos *dom1* e *dom2*, consiste de associar variáveis de domínio ocorrendo em *dom1* para seus correspondentes estruturais em *dom2*. Os domínios *dom1* e *dom2* são unificáveis se e somente se as seguintes condições são satisfeitas:

1. A estrutura de *dom1* é mais geral que a de *dom2*, isto é, as variáveis de domínio em *dom1* podem corresponder às expressões de domínio em *dom2* mas não vice-versa.

2. Todo identificador de domínio em *dom1* corresponde estruturalmente a um domínio em *dom2* e estes são fortemente equivalentes.

3. Ocorrências distintas de uma mesma variável de domínio em *dom1* devem corresponder a domínios fortemente equivalentes em *dom2*.

4. Se *dom1* e *dom2* são uniões de domínios simples então cada operando de *dom2* deve ser unificável com um operando de *dom1*.

Note que a proposta de unificação é unilateral, isto é, ela unifica o primeiro domínio com o segundo. Somente as variáveis de tipo do primeiro domínio são associadas com expressões de tipo do segundo domínio. Esta forma de funcionamento do mecanismo de unificação está de acordo com o meio como é feita a verificação de tipos em SDL.

4.2.4 Particularização

Considere a seguinte expressão SDL e as especificações de domínio associadas:

$\mathtt{s} := @1 \rightarrow @1;$
$\mathtt{t} := N \rightarrow N;$
LET $\mathtt{s} = \mathtt{t}$ IN x.

[7] Ver seções 3.3 e 3.4.

A regra 7 que descreve o sistema de tipos da linguagem SDL estabelece que o tipo de `t` deve ser ou igual ou uma particularização do domínio de `s`. Ainda, o tipo de `s` é esperado ser $N \rightarrow N$ no escopo de `x` por causa do *binding* indicado. Em resumo, o tipo de `s` deve ser unificado com o de `t` para produzir o domínio local de `s`. Deve ser implementada uma função que descreve formalmente este mecanismo.

5 Avaliação do Sistema

O algoritmo de equivalência que foi aqui mostrado revelou-se bastante poderoso e bem flexível. Até o momento ainda não foi elaborada uma prova de completeza e de correção das regras estabelecidas. Entretanto, a implementação desenvolvida mostrou que os casos esperados de polimorfismo foram tratados adequadamente pelo sistema.

Cabe ainda observar que Milner [11] provou que, com este sistema de tipos, podem existir expressões que seriam consideradas mal-tipadas embora pareçam perfeitamente normais. Com o objetivo de provar provar este ponto, Milner apresentou como exemplo a expressão LET `g(f)` = λ `(a,b) . (f(a),f(b))` que é definida no escopo das especificações

$$\begin{aligned}
&\texttt{f} := @1 \rightarrow @1;\\
&\texttt{a} := @1;\\
&\texttt{b} := @1;\\
&\texttt{g} := (@1 \rightarrow @1) \rightarrow (@3, @4) \rightarrow (@3, @4);
\end{aligned}$$

Se a substituição dos domínios acima produz os tipos

$$\begin{aligned}
&f := @12 \rightarrow @12\\
&a := @10\\
&b := @11
\end{aligned}$$

então o tipo de `f(a)` é @12 sendo @12 um sinônimo para @10. O tipo de `f(b)` é também @12 mas agora @12 é sinônimo para @11. Desde que uma mesma variável de domínio não pode ser associada a dois valores diferentes ao mesmo tempo então a expressão acima é rejeitada pelo sistema de tipos.

Basicamente a expressão acima é mal-tipada porque viola a regra 2. De certa forma isto não é um resultado completamente inesperado porque o sistema de tipos foi forçado a adotar restrições que reduzem o grau de polimorfismo. Estas restricões ocorrem em contextos particulares tal como no caso de parâmetros formais, argumentos para abstrações λ e etc. Isto é certamente uma limitação na flexibilidade do sistema de tipos porque parâmetros e variáveis de abstrações não apresentam a mesma liberdade de outras variáveis de tipo. Contudo ainda não é claro o quanto restritivo isto é na prática e a questão de como o sistema de tipos deveria ser extendido para cobrir estes casos particulares está em aberto.

6 Conclusão

O sistema **xlds** [2, 9, 1, 4, 13] encontra-se ainda em fase de testes e o módulo que vai executar o código LAMB que é gerado está ainda em fase implementação. No entanto, toda a verificação de tipos que é executada pelo compilador SDL já está disponível e o retorno por parte da comunidade de usuários vai ser útil na determinação dos limites que foram comentados quanto ao poder deste sistema de tipos.

Referências

[1] Walace A. Rodrigues. Compilação e otimização de uma linguagem para definição denotacional de semântica. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte - MG, 1993.

[2] L.M. Amaral. Interface homem-máquina do ambiente de definição semântica LDS. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte - MG, 1992.

[3] D.M. Berry. Type equivalence in strongly typed languages: One more look. Technical report, Faculty of Mathematics of Weizmann Institute, Israel, 1978.

[4] R.S. Bigonha. *A Denotational Semantics Implementation System.* PhD thesis, University of California, Los Angeles, 1981.

[5] D.S.Scott C.A.Gunter and P.D.Mosses. Semantics domains and denotational semantics. Technical report, University of Aarhus, Denmark, 1989.

[6] Luca Cardelli and Wegner Peter. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), Dec. 1985.

[7] M.J.C. Gordon. *The Denotational Description of Programming Languages - An Introduction.* Springer-Verlag, New York - Heiberg - Berlin, 1979.

[8] C.A. Gunter. Semantics of programming language - structures and techniques. *MTI Press*, 1992.

[9] J. Leite S.Jr. Linguagem de definição e geração de analisadores sintáticos em semântica denotacional legível. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte - MG, 1993.

[10] A.R. Milne and C. Strachey. *A Theory of Programming Language Semantics*, volume a and b. Chapman and Hall, London, 1976.

[11] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[12] P.D. Mosses. Sis – a compiler-generator system using denotational semantics. Technical report, University of Arhus, 1978.

[13] W.A.Rodrigues R.S.Bigonha, J.S.Leite and L.M.Amaral. Semântica denotacional legível. In *Seminário Informática 25*, Pontifícia Universidade Católica, Rio de Janeiro - RJ, agosto 1992.

[14] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata*. Polytechnic Institute of Brooklyn, 1971.

[15] D.S. Scott. Outline of a mathematical theory of computation. Technical monograph prg-2, University of Oxford, 1971.

[16] M. Solomon. Type definitions with parameters. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, University of Wisconsin – Madison, Jan. 1978.

[17] J.E. Stoy. Denotational semantics: The Scott-Stratchey approach to programming language theory. *MTI Press*, 1977.

[18] J.E. Stoy. Foundations of mathematical semantics. Lecture Notes, Copenhagen, Jan. 1979.

[19] R.D. Tennent. Language design methods based on semantic principles. *Acta Informática*, 8(2):97–112, 1977.

[20] R.D. Tennent. Another look at type compatibility in pascal. *Software – Practice and Experience*, 8:429–437, 1978.

[21] J. Welsh. Ambiguities and insecurities in pascal. *Software – Practice and Experience*, 7:685–696, 1977.