COINS プロジェクト LIR 仕様書

2002 年 7 月 27 日

目 次

1　導入

LIR は COINS プロジェクトの低水準中間表現の名称であり、Low-level Intermediate Representation の略である。以後本ドキュメントでは中間表現ではなく、中間言語という言葉を使う。LIR はコード最適化から最終的な機械語生成までの各フェーズにおいて共通に使われる中間言語である。

この導入セクションの目的は LIR の設計目標と設計思想を説明することであり、LIR の正式な定義は含まないので飛ばしても差し支えない。

1.1　LIR の設計目標

中間言語に対する要求事項は多いが、あまり詳細な項目を定めても繁雑になるばかりだし、それは当然の要求を書き連ねただけの無意味なものになる。我々は中間言語にとって本質的であり、かつ方向性の異なる目標として以下を提示する。

1. 豊かな表現力
2. 豊かな理論
3. 仕様の簡潔さ

LIR はさまざまなソース言語、さまざまなターゲットマシンにおいて使用出来るものでなければならない。従って豊な表現力が必要とされる。またコンパイラのパスは中間言語を静的に解析し、変換していくのだから、中間言語は単にプログラムを表現出来ればよいというだけでなく、静的な解析のよりどころとなる定理の集合としての理論が豊かでありたい。例えば中間言語で表現された変数について、明示的な代入がないかぎりその値は変更されない、という定理は最適化で当然期待したい定理であろう。簡潔であることは中間言語にかぎらず全てのソフトウエアに共通の要求であるが、コンパイラのような複雑でかつ高い信頼性を求められるソフトウエアでは本質的である。

これらの目標達成度の計量をフォーマルに定義することは出来ないが、中間言語の表現力とはそれが表現出来る命令の集合、理論とは静的解析のための定理の集合で、ともに包含関係に関して大きいほうが豊かであると考えられる。また簡潔さとはそのドキュメンテーションの文字数の少なさと考えて良いかも知れない。我々の目標をより明確にするため以下に極端な例を上げる。

豊かな表現力のみ満足する中間言語の例として、ターゲットマシンのアセンブラ言語そのものが考えられる。どのような高級言語であれ、結局ターゲットマシンの機械語に変換されるのだから、表現力の点では究極といえる。またアセンブラ言語のニーモニックの衝突はターゲットマシンの名前をプリフィックスに付けるなどして解決出来るので、任意のマシンを表現することは可能である。いうまでもなく、このアプローチは仕様の簡潔さを欠く。また、制限のない所に理論は生まれない、という原理どおり、アセンブラ言語は表現力が豊かすぎるので、理論は貧弱になり、手続きとか基本ブロックというコード最適化で重要な概念も存在出来ない。

豊かな理論と仕様の簡潔さを合わせ持つ表現としてはラムダカリキュラスがある。たしかにラムダカリキュラスは表示的意味論の言語としても使えるのだから、そういう意味では任意のプログラミング言語や任意のマシン命令を「表現出来る」ので、豊かな表現力も持つと言えるかもしれない。しかしその表現はコンパイラのパスが扱うのに適したものではない。すなわち我々の目標にいう「表現」とは単に表現出来ればよい、という以上に、その意味を自然に表現出来なければならない。

この「自然に表現出来る」という要求は我々が提示する目標に含まれていないのではないか、それならば「自然に表現出来る」という項目を、「プログラムをその中間言語で表現した場合の簡潔さ」と定義し、目標に加えるべきではないか、という意見が出そうである。実際「自然に表現出来る」ということは中間言語の設計で重要な点であり、そうでなければラムダカリキュラスや帰納的関数などといった連中が候補として名乗りを上げてくるかもしれない。

しかし、「自然に表現出来る」ということを安直に上のようには定義できないのである。後に設計思想のところで提示する我々の案は上の定義は満たさないが、自然な表現と考えられる。我々はこの目標を直接定義するのではなく、多少詭弁的ではあるが、次のように考えることで、LIR の設計目標を上のみっつにまとめることにする (目標自体の簡潔さも重要である)。

我々は LIR を正式なプログラミング言語として設計する。この理由については後に説明する。プログラミング言語である以上、その仕様はシンタックス、セマンティックス、及びプラグマティックスから成り立つ。プラグマティックスとは言語の使い方であり、中間言語の場合にそれはソース言語から LIR への変換、コンパイラのパス構成、ターゲットマシン命令の表現方法などを意味する重要な物となる。すなわち我々の目標にいう仕様の簡潔さはこれらの簡潔さも含むとするのである。したがって、ラムダカリキュラスや帰納的関数は除外される。

1.2 LIR の設計思想

ここでは LIR の設計がどのような考えでなされているかを説明する。LIR はコード最適化から最終的なコード生成、加えてピープホール最適化やパイプラインスケジューリングまでの各フェーズで使われる表現であることから、少なくともターゲットマシン命令を表現できなければならない。またそれが表現出来るということはソース言語も表現出来ることになる。よって LIR は基本的にターゲットマシン命令を表現する中間言語として設計する。高級言語に存在するような型システムは LIR には不要であり、LIR が区別する型は実際のマシンが区別する型のみである。

ここで解決しなければならない問題は先に提示したみっつの目標を同時に満たす解決策を考えることであり、部分的に無理であっても、そこに対して多くの人がなっとく出来るような妥協点を見出すことである。

まず第一に、さまざまなマシンの命令を表現することが問題である。第二に、豊かな理論を提供することが問題である。以下に述べるように、第一の問題についての基本的なアプローチは現在すでに確立されているし、我々もそれに従う。しかし第二の問題については、そのコンパイラの目的、対象言語などに依存したさまざまな選択が考えられる。我々の提案は COINS プロジェクトの目的を考慮しつつ捻出したひとつの妥協案にすぎない。

1.2.1 第一の問題

まず対象とするマシンであるが、現存する全てのマシンを表現することはあきらめる。それは他の目標の義性なしには不可能だからである。我々は COINS が対象とするマシン、すなわち複数の汎用レジスターを持つ普通の RISC 及び CISC に対象をしぼり、そこから設計を始めていき、理論と簡潔さの顔色をうかがいつつ、対象とするマシンを広げていく予定である。

次にさまざまな命令を少ない種類の中間言語で表現する方法について述べる。まず先の極端な例で上げた、ターゲットマシンのアセンブラコードを中間言語とする、という案は論外であるが、同様の提案が過去になされ、長く議論もされたのでふれておく。このアプローチではコンパイラの各パスは命令数の数だけ場合分けをしなければならない。よってターゲットがごく小規模の RISC である場合を除いて、そして特に多様なアドレッシングモードを持つ CISC において、その場合分けの数は膨大なものになってしまう。

一般にマシン命令はその命令の本来の演算以外に、事実上アドレッシングモードとしてアドレスの加算やメモリ参照が行なわれる。よって表現の最小単位を命令そのものとするのではなく、より基本的な演算からなる基本セットを作り、各マシン命令はその基本セットを組み合わせて表現すればよい。もう少し具体的にいえば、それらの基本セットをノードとするツリーで表せばよいのである。その基本セットとして必要なのは従って命令本来の演算に加えて、各アドレッシングモードで暗黙的に行なわれる演

算からなる集合であり、命令全体よりはるかに小さい。

複雑な命令をこの基本セットで表現した場合、その複雑な命令を一つの命令と扱うのにくらべて、たしかに見ためは複雑である。しかしその表現は命令の意味をより基本的なセットで表現しているのであり、コンパイラのパスはその基本セットについての知識のみあればよい。一方複雑な命令をそのまま表現として使うということは、その複雑な意味を各パスが全て知っていなければならないのである。

いうまでもなく、中間言語をツリーで表現しマシン命令を表すツリーのパターンマッチによりコード生成を行なうという構成は既にリターゲッタブルコンパイラの常識であり [1]、GCC [3] もこのアプローチで成功している。このアイディアは既に古典であるが、自然で強力であり、我々もこのアプローチを採用する。

さらに、LIR には GCC の中間表現である RTL から拝借した命令もある。実際 GCC のような優れた移植性をもち、成功しているコンパイラから学ぶことは多い。しかし我々は無批判にただ GCC の RTL を拝借したわけではない。さまざまな議論を経て、結果として類似のものに至ったのである。設計過程の議論において、必然的に必要となる特殊な命令で、GCC でも存在する同様の命令については、その命令の名称をあえて GCC のものと変更することはしなかった。議論の過程で GCC を 参考にしたことは事実であるし、機能的にも同一のものであるならば、むしろ GCC に敬意を表する意味でも、その名前を使うべきである。

1.2.2 第二の問題

第一の問題が解決したとして、それだけでは結局アセンブラ言語を表現出来ているに過ぎない。コンパイラにとって都合のよい理論を得るためには中間言語に対して何らかの制限が必要である。しかし制限を加えることは良質なコード生成の妨げになる。ここで制限というのはアセンブラ言語の使い方を制限するということで、高級言語で例えて言えば勝手な goto 文を禁止し、より制限された goto 文である if 文や while 文を導入するということである。

大雑把に言えば、制限なし、すなわちアセンブラ言語そのままという状態では最適化のインプリメントは難しい、しかし制限がないのだから可能な限り良いコードを出すことは努力しだいでは可能である。一方で制限を多くして理論を豊かにすると、最適化のインプリメントは容易だがその制限により不可能となる最適化については、あきらめるよりない。すなわちこの第二の問題はトレードオフの問題であり、総合的な見地から何らかの妥協点を見出すという問題である。

GCC は前者寄りのアプローチである。そして我々は後者寄りのアプローチを試みる。これはパス構成とも関連しており、詳しくは LIR のプラグマティックスの所で説明されるが、ここでは GCC と比較する形で概略を説明する。

GCC は RTL 生成時に既にパターンマッチングを終了しており、RTL はターゲットマシンの実在する命令を表している。 以後最適化フェーズの各パスは基本的にその形を崩さないように配慮しつつ RTL を変換ていく。これは確かに正直なアプローチであり、命令に応じた細かい最適化を適切に行なえる。一方でこのために各パスのインプリメントは繁雑になる。関数呼びだしなどでは特定の物理レジスタが RTL の始めから登場する。これをケアするのは注意が必要である。物理レジスタは仮想レジスタと違い、さまざまな制約があるからである。

我々はマシン命令を表現するためのコード (L 式) に加えて LIR により上位の構造を導入する。これらの構造はアセンブラ言語の使い方を制限するが、代償としていくつかの理論が手にはいる。

1. L 式
2. L 関数
3. L モジュール

4. L プログラム

L 関数は構文上は L 式の列に関数としてのインタフェースを付加したものである。L 関数は一般に多値を返す、副作用のある関数を表す。これとデータが合わさり L モジュールを構成する。これがひとつのコンパイル単位である。これらの外部参照を解決するような集まりが L プログラムである。詳細は LIR のプラグマティックスで述べるが、これらの導入により例えばコード最適化フェーズでは仮想レジスタについて以下を仮定することが出来るようになる。これは LIR の理論が提供するひとつの定理である。

1. 名前の任意性
   レジスタの名前を任意に変更してもプログラムは等価である。

2. 独立性
   あるレジスタに代入した際、他のレジスタは破壊されない。

3. 代入の明示性
   レジスタに対する代入はプログラム中に明示的に現れる。

4. 使用の明示性
   レジスタに対する使用はプログラム中に明示的に現れる。

これらの定理はコード最適化では当然仮定したいが、GCC では幾つかのレジスタが物理レジスタであり、これらの条件全ては満たされない。よって個別にケアが必要とされるのである。我々のアプローチでは L 関数という構造により、関数呼びだしに関係する物理レジスタの割り当てを後に遅らせることが出来る。

おそらく、コードのクオリティを追求するのであれば GCC のアプローチがすぐれているかもしれない。しかし COINS の目標のひとつは新たな技術を試すための基盤を提供することであり、コード最適化フェーズが書きやすいというのは考慮すべき点である。コード最適化が書きやすいということはより高度な最適化を導入することが容易となる。また L 関数の導入により、GCC 的アプローチよりも柔軟なインライン展開、部分評価が可能となる。我々のアプローチに対する今後の評価を期待したい。

1.2.3 プログラミング言語 LIR

コンパイラの中間言語は日陰の存在であった。特定言語、特定機種のカリカリに高性能なコンパイラであればそれでよいし、企業で作られるコンパイラなどではむしろ隠さなければならない、という状況も考えられる。しかし COINS の目的上、明らかに LIR は人前にさらされなければならない。メモリー中に存在するただのデータ構造であってはならないのである。

例えば C 言語を説明するのに、ある特定の C コンパイラのデータ構造やアクセスメソッドの仕様により C の言語仕様を説明する人はいない。もし C 言語というものが、単にあのようなシンタックスをもつデータ構造であるなら、その方法でも意味は通じるかも知れないが C 言語は実際にはセマンティックスを伴った対象であり、重要なのはむしろセマンティックスである。そのセマンティックスを定義する対象であるべきシンタックスオブジェクトが、ある C コンパイラのデータ構造といった不必要に具体的な対象である必要性は何もない。

中間言語が具体的なデータ構造として定義されてきた大きな理由は中間言語の性質上、それをインプリメントする言語が決まっているという特殊な状況に由来すると思われる。この状況は COINS の LIR でも同じである。にもかかわらず、中間言語を独立したプログラミング言語として設計することは以下に述べるように現実的な意味で意義があるのである。

我々は中間言語 LIR を正式なプログラミング言語として設計する。すなわちそのシンタックス、セマンティックス、及びプラグラティックスを与えることで LIR を定義する。具体的なインプリメントは独立したドキュメントとなる。これにより LIR の仕様をインプリメントと完全に分離することが出来、インプリメントと独立に LIR の議論や研究が可能となる。また、より良いインプリメントに変更する場合でも、LIR の仕様書は一切変更する必要がない。データ構造としてのみ定義されている中間言語は、とかく無節操に複雑なリンクを持ちたがる。加えて基本ブロック、フロー情報、ループ情報といった、与えられた中間言語から生成出来るような情報、果ては特定のパス内部でのみ一時的に使う情報なども中間言語の一部として語られてしまう。LIR のシンタックスをきちんと定義しなければならないとすると、本質的でないものは捨てて、可能な限り簡潔明解にしたいという努力が自然に生まれる。また本質的なものと本質的でないものが明確になり、LIR を理解することが容易になる。LIR のリーダとプリンタは容易に作成出来る。LIR の L モジュールはひとつのコンパイル単位であり、そのシンタックスに従って例えばファイルにセーブされたあとで、後に完全に良み戻してコンパイルを継続することが出来る。これはバックエンドの簡潔なインタフェースを提供することになる。LIR の具体的なインプリメントを理解しなくても LIR のシンタックスのみ理解することで COINS のバックエンドを利用することが可能となり、特に COINS のインプリメント言語以外の言語でパスをインプリメントすることが可能となる。まとめると以下のとおり。

1. LIR が簡潔になる
2. LIR の理解が容易になる
3. バックエンドの簡潔なインタフェースを提供する

次にセマンティックスの定義について述べる。我々は LIR のセマンティックスを表示的意味論により厳密に定義する。 どんな言語であれ、言語仕様は厳密であるに越したことはないが、リターゲッタブル性を考慮した中間言語において厳密な定義は本質的である。とくに COINS ではつぎの理由も上げられる。

COINS のインプリメント言語 Java は命令の仕様が厳密に決められており、どのマシンで動く Java も同じ命令は同じ意味を持つ。したがってあるターゲットマシン上で、そのマシンのオブジェクトコードを生成する、という状況においてすら、COINS コンパイラは本質的にクロスコンパイラである。すなわち Java のある演算は、一見対応するように見えるターゲットマシンでの演算と振舞いが異なる可能性があり、典型的には定数畳み込みなどでバグの原因となりかねない。このような危険性を回避するために命令の意味を厳密に定義する必要が生じる。

命令の意味を厳密に定義するためにはメモリというハードウエアも厳密に定義する必要が出てくる。また L 関数や L モジュールという上位構造の意味も定義しなければならない。その結果、LIR という言語の実行モデルが定義されるだけでなく、LIR の厳密な実行系を作ることが出来る。それは部分評価のような最適化やコンパイラのデバッグに役立つと思われる。

また COINS コンパイラはマシン記述言語に基づいたリターゲッタブルコンパイラを目指しているが、セマンティックスの定義に現れるマシン依存関数はマシン記述言語の設計において、信頼出来る土台となる。マシン記述言語の意味とはそれらのマシン依存関数の定義、及びマシン命令と L 式の対応として定義される。

LIR のセマンティックスが厳密に定義されるということは、各パスが行なってよいプログラム変換が厳密に定義出来るということを意味する。すなわち行なって良い変換とは中間言語の意味を変えない変換である、というあたりまえのことが、厳密に定義出来る。もちろん、これからコンパイラの自動検証が可能である、という大胆な主張はしないし、それは COINS プロジェクトの目的でもない。しかしセマンティックスの厳密な定義は、そのような方面の研究者のみならず、理論的なアプローチを重要視するコンパイラ研究者にもアピール出来るものとなる。そして、多くの研究者が思わず使ってみたくなるような魅力的なインフラを提供することは COINS プロジェクトの根本的な目標である。

2 準備

ここでは本ドキュメントで使用する表記法及び前提としたい予備知識をまとめる。ここでは本ドキュメントでのみ使われる特別な記法やことばの定義もなされるので、最初にここをかならず読まれたい。ここで引用している一般的な概念についての詳しい説明は省略する。それらについては必要なら付録の参考文献を参照されたい。

2.1 表記法のまとめ

定義を意味する記号は 「$\stackrel{\triangle}{=}$」を使う。例えば $\mathtt{foo} \stackrel{\triangle}{=} \{1, 2, \ldots, 100\}$ は foo を 1 から 100 までの自然数の集合であると定義している。このように、常識的に判断して誤解がない範囲で、表記の簡潔さのために「$\cdots$」を使用する。

論理記号は $\neg,\ \wedge,\ \vee,\ \rightarrow,\ \forall,\ \exists,\ \exists!$ を使う。 $\exists! x \in X\ P(x)$ は $P(x)$ である $x \in X$ がただ一つ存在することを意味する。ことのきその一意的に存在する x を $!x \in X\ P(x)$ で表す。

論理記号の構文的な結合順位は最初に提示した順に低くなるとする。他の集合や演算の記号についても一応はこのように結合順位を定めるが、誤解の恐れがある場合には冗長な括弧により誤解の無い記述を行なう。

集合論の表記 $\emptyset, \{x_1, \ldots, x_n\},\ \{x \in X \mid P(x)\},\ A^B,\ \times,\ \cup,\ \cap,\ +,\ -,\ !,\ \subset,\ \in$ を使う。二項記号の結合順位はこの順で低くなるとする。$\emptyset$ は空集合を表す。集合の包含関係「$\subset$」は広義、すなわち $A \subset B \stackrel{\triangle}{=} \forall x \in A\ x \in B$ である。$A \times B$ は直積、A^B は配置集合であり、B が自然数 n の場合は A の n 直積 $A \times \cdots \times A$ とする。直積 $A_1 \times \cdots \times A_n$ の要素は $x_i \in A_i$ として $(x_1, \ldots, x_n)$ で表す。

本ドキュメントにおいて直積の要素や後に定義するリストを含め、一般にある列の要素を特定する際、「n 番目」 という表現は 0 オリジン $(0 \leq n)$、「第 k」という表現は 1 オリジン $(1 \leq k)$ とする。例えば

> 列 a, b, c について、
> a は 0 番目の要素、b は 1 番目の要素。
> a は 第一要素、b は第二要素。

直積の要素 $x = (x_0, x_1, \ldots)$ の i 番目の要素 x_i を $x!i$ で表す。$A_1 + \cdots + \mathcal{A}_n$ は内部直和である。すなわちそれは $A_1 + \cdots + A_n = A_1 \cup \cdots \cup A_n$ でかつ $i \neq j \rightarrow A_i \cap A_j = \emptyset$ なる集合である。外部直和は使用しない。$A - B$ は集合の差、すなわち $A - B \stackrel{\triangle}{=} \{x \in A \mid x \notin B\}$ である。$\cup, \cap$ は $\cup_{i=1}^n S_i,\ \cap_{i=1}^n S_i$ という形でも使用する。($\cup_{i=1}^n S_i \stackrel{\triangle}{=} S_1 \cup \cdots \cup S_n$ である。$\cap_{i=1}^n S_i$ についても同様である。)

本ドキュメントでは以下の型及び型構成子を使用する。構成子の結合順位はこの提示順に低くなるとする。直積、直和、差は集合のそれと同じであるが、構成子全体の結合順位明記のために加えてある。型は単純に集合と扱う。よって型の等価性は集合のそれである。型と呼ぶか集合と呼ぶかは単に気分の問題である。同様に $a{:}\,X$ というシグネチャは $a \in X$ と同じであり、使い分けに深い意味は無い。

$\boldsymbol{R}$ は実数の集合であるが、通常の意味で順序体 $(\boldsymbol{R}, \leq)$ とも見なす。一般に空でない全順序集合 $(X, \leq)$ について、その最大値を $\mathtt{max}\, X$、最小値を $\mathtt{min}\, X$ で表す。$\boldsymbol{R}$ の体としての四則演算は $*, /, +, -$ で表す。結合順位は通常通りである。 また $-$ は単項のマイナスの意味でも使い、結合順位は二項演算より高い。 $\sum$ は総和記号を表す。$x, y \in \boldsymbol{R}, 0 < x$ について x^y で冪乗を表す。

これらの演算は $\boldsymbol{R}$ の部分集合にも適用する。例えば $a, b \in \boldsymbol{Z}$ について、a/b は $a, b \in \boldsymbol{R}$ と見なした演算であり、$a/b \in \boldsymbol{R}$ である。

本ドキュメントでは乗算記号は省略しない。したがって ab は a と b の積ではなく、ab という記号であるが、ラムダ記法の束縛変数は例外である (後述)。

$$
\begin{aligned}
\boldsymbol{N} &\triangleq 0\text{ 以上の整数の集合}\\
\boldsymbol{Z} &\triangleq \text{整数の集合}\\
\boldsymbol{R} &\triangleq \text{実数の集合}\\
\boldsymbol{Bool} &\triangleq \{\mathbf{true}, \mathbf{false}\}\\
\boldsymbol{Type} &\triangleq \text{自身を除く本ドキュメントに現れる全ての型の集合}\\
[\tau] &\triangleq \tau\text{ のリスト}\\
\{l_1{:}\,\tau_1, \ldots, l_n{:}\,\tau_n\} &\triangleq \text{ラベル } l_i \text{ の型が } \tau_i \text{ であるレコード型}\\
\tau_1 \times \cdots \times \tau_n &\triangleq \tau_1, \ldots, \tau_n \text{ の直積}\\
\tau_1 + \cdots + \tau_n &\triangleq \tau_1, \ldots, \tau_n \text{ の直和}\\
\tau_1 - \tau_2 &\triangleq \tau_1 \text{ と } \tau_2 \text{ の差}\\
\alpha \to \beta &\triangleq \alpha \text{ から } \beta \text{ への関数型 } (=\ \beta^\alpha)\\
a{:}\,\alpha \to \beta(a) &\triangleq \text{依存型 (ここに } \beta{:}\,\alpha \to \boldsymbol{Type})
\end{aligned}
$$

整数に関してガウスの合同式を使用する。

$$x \equiv y\ (\mathtt{mod}\, m) \triangleq x, y, m \in Z \wedge \exists k \in \boldsymbol{Z}\ m * k = x - y$$

実数を整数に変換する以下の関数を使用する。

$$
\begin{aligned}
\mathtt{floor}, \mathtt{ceiling}, \mathtt{truncate} &: \boldsymbol{R} \to \boldsymbol{Z}\\
\mathtt{floor}\, x &\triangleq \mathtt{max}\{z \in \boldsymbol{Z} \mid z \le x\}\\
\mathtt{ceiling}\, x &\triangleq \mathtt{min}\{z \in \boldsymbol{Z} \mid x \le z\}\\
\mathtt{truncate}\, x &\triangleq \mathbf{if}\ 0 \le x\ \mathbf{then}\ \mathtt{floor}\, x\ \mathbf{else}\ \mathtt{ceiling}\, x
\end{aligned}
$$

これらは実数をそれに近い整数に変換する。`floor` は $-\infty$ に向かい、`ceiling` は $+\infty$ に向かい、`truncate` は 0 に向かって最も近い整数を対応させる。

条件式 **if** c **then** x **else** y は $c \in \boldsymbol{Bool}$ が **true** なら x、**false** なら y を表す。 条件式は c について strict だが x, y については non strict とする。$\exists x\, P(x) \to \exists! x\, P(x)$ なる述語について、条件式 **if** $\exists! x\, P(x)$ **then** $f\, x$ **else** y は一意的に存在したら $f\, x$ さもなくば y を表す。またケース式 **case** e **of** $l_1 \Rightarrow v_1 \cdots l_n \Rightarrow v_n$ も使用する。これは $e = l_i$ ならば v_i を表す。ケース式は e, l_i について strict、v_i について non strict である。 ケース式を使う場合は $\exists! i\ e = l_i$ でなければならない。

関数型の型式は右結合である。すなわち $\alpha \to \beta \to \gamma$ は $\alpha \to (\beta \to \gamma)$。関数適用 $f(x)$ は必要なければ括弧を略して $f\ x$ と書く。関数適用は左結合である。すなわち $f\ x\ y = (f\ x)\ y$。結合順位については、関数適用が最も結合度が高く、次に単項演算子 (形式上関数と同様引数の左に置かれるが、英数字以外の記号で構成されるもの)、最後に二項演算子の順である。すなわち $f\ x * -\ g\ x\ y = (f\ x) * (-((g\ x)\ y))$ 関数 f の定義域を $\mathtt{dom}\, f$ で表す。また関数 f の $X \subset \mathtt{dom}\, f$ への制限を $f|_X$ で表す。

$\lambda x.M$ はラムダ記法であり、$f(x) \triangleq M$ なる関数 f を表す。これはラムダカリキュラスのラムダ式ではない (本ドキュメントではラムダカリキュラスは使用しない)。これは主に演算子のシグネチャを与えるために使用されるので、簡潔さのために慣例に合わせて束縛変数は 1 文字であるとする。つまり $\lambda xy.M = \lambda x.(\lambda y.M)$。

関数引数は一般に括弧を略すが、意味記述において、意味関数定義の関数引数及び定義右辺に現れるシンタックスオブジェクトは $[\![\ ]\!]$ で囲む。これは意味関数ではシンタックスオブジェクトとセマンティックスオブジェクトが入り乱れるので、定義中のどこがシンタックスオブジェクトかを明示するためだけ

の慣習である。とくに本ドキュメントに限って言えば、シンタックスオブジェクトを S 式で表現している都合上、例えば $\mathcal{E}[\![(\texttt{CONVSX}\, t\, x_1)]\!] \triangleq \cdots$ なる記述において、$[\![\,]\!]$ がないと混乱の恐れがある。意味関数及びその補助関数以外では原則としてシンタックスオブジェクトでも $[\![\,]\!]$ で囲まないが、使い分けは多分に気分の問題である。要は混乱をさけるためだけの記法であることを理解されたい。

ローカルな定義を行なうために「式 **where** 定義列」という表記を使用する。この場合定義列で定義される記号のスコープはこの表記内である。以下は例である。

$$\begin{aligned}\texttt{big} &\triangleq f\,100 \quad \textbf{where}\\ f &: \boldsymbol{N} \to \boldsymbol{N}\\ f\,0 &\triangleq 1\\ f\,n &\triangleq n * f\,(n-1) \quad \textbf{if } n \neq 0\end{aligned}$$

この f の定義は引数により場合分けをした定義の例にもなっている。通常数学で行なわれるこのような定義法も使用する。定義右辺にある「**if** 条件」は条件が満たされる場合のみその定義が使われることを意味する。

依存型は引数の「値」に依存して結果の型が決定する関数の型を表す場合に使用する。以下は依存型をもつ関数の定義例である。

$$\begin{aligned}\texttt{zeros} &: n{:}\,\boldsymbol{N} \to \boldsymbol{N}^n\\ \texttt{zeros}\,n &\triangleq (\underbrace{0,\ldots,0}_{n\text{ 個}})\end{aligned}$$

X に $\bot_X$ を追加した集合を $X_\bot$ で表す。混乱がないならば $\bot_X$ は $\bot$ と略す。 $X_\bot$ は flat domain $(X_\bot, \sqsubseteq)$ と考える。部分関数 $f{:}\,X \to Y$ は全関数と flat domain により $f{:}\,X \to Y_\bot$ という形で表す。すなわち $\forall x \in X\ x \in \texttt{dom}\, f \leftrightarrow f\,x \neq \bot$ である。関数空間のオーダー $\sqsubseteq$ は通常どうり $f \sqsubseteq g \triangleq f\,x \in \texttt{dom}\, f \ \rightarrow\ f\,x = g\,x$ で定義する。意味定義で現れる関数とコンストラクタは特に断らない限り全てストリクトであるとする。すなわち $f{:}\,X \to Y$ という関数定義には $f(\bot_X) \triangleq \bot_Y$ という定義が付随しているとする。関数型 $\tau \triangleq \alpha \to \beta_\bot$ について、$\bot_\tau$ は $\forall x \in \alpha\ \ fx = \bot_\beta$ なる関数 f とする。

リスト型 $[\tau]$ は後に説明する S 式のリストとは別ものである。要素は全て同一の型でなければならない。リスト型 $[\tau]$ のコンストラクタも [] で表す。すなわち $x_1,\ldots,x_n\ (x_i \in \tau)$ のリストは $[x_1,\ldots,x_n] \in [\tau]$ で表す。リスト演算として以下を使用する。

$$\begin{aligned}\# &: [\tau] \to \boldsymbol{N}\\ \#[x_1,x_2,\ldots,x_n] &\triangleq n\\ \lambda xn.(x\,!\,n) &: [\tau] \to \boldsymbol{N} \to \tau_\bot\\ x@[x_0,x_1,\ldots,x_n,\cdots]\,!\,n &\triangleq x_n \quad \textbf{if } n < \#\,x\\ x\,!\,n &\triangleq \bot \quad \textbf{if } n \geq \#\,x\\ \lambda ex.(e \in x) &: \tau \to [\tau] \to \boldsymbol{Bool}\\ e \in [x_0,x_1,\ldots] &\triangleq \exists x_i\ \ e = x_i\\ \lambda xy.(x +\!\!+\, y) &: [\tau] \to [\tau] \to [\tau]\\ [x_1,\ldots,x_n] +\!\!+\, [y_1,\ldots,y_m] &\triangleq [x_1,\ldots,x_n,y_1,\ldots,y_m]\\ \texttt{reverse} &: [\tau] \to [\tau]\\ \texttt{reverse}\,[x_1,\ldots,x_n] &\triangleq [x_n,\ldots,x_1]\end{aligned}$$

$\#\,x$ でリスト x の長さを表す。定義の左辺にはこのようにパターンを使用する。定義左辺の引数パターンにおいて *italic* 体で書かている変数にマッチしたものが右辺で参照される。$x\,!\,i$ はリスト x の i

番目の要素を表す。++ はリストの結合、reverse は反転である。ここで定義左辺 「x@パターン」の x は定義の右辺で「パターン」がマッチしたもの全体を表す。★パターンバインディングの説明。

関数 $f\colon X \to Y_\perp$ は集合論における通常の解釈に従い直積 $X \times Y$ の部分集合と考える。空集合は従って関数型の $\perp$ である。ただしその直積の要素は分かりやすさのために $x \mapsto y$ で表す。例えば $f\,x_i \triangleq y_i\ i \in I$(インデックス集合) なる部分関数 $f\colon X \to Y_\perp$ は $\{x_i \mapsto y_i \mid i \in I\}$ という集合で表すことも出来る。 インデックス集合が明らかな場合は単に $\{x_i \mapsto y_i\}$ とも書く。

関数 $f\colon X \to Y_\perp$ について、$f := \ x_1 \mapsto y_1$ により、$g\,x = \textbf{if}\ x = x_1\ \textbf{then}\ y_1\ \textbf{else}\ f\,x$ なる関数 $g\colon X \cup \{x_1\} \to Y$ を表す。また $\sigma_i \triangleq x_i \mapsto y_i$ のとき $f := [\sigma_1, \ldots, \sigma_n] \triangleq ((f := \sigma_1) := \sigma_2) \cdots := \sigma_n$ とする。また関数 $f, g\colon X \to Y_\perp$ について、$f := g$ により、$h\,x = \textbf{if}\ g\,x \neq \perp\ \textbf{then}\ g\,x\ \textbf{else}\ f\,x$ なる関数 $h\colon X \to Y_\perp$ を表す。

レコード $r\colon \{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}$ について、$r := \{.l_{n_1} = v_{n_1}, \cdots, .l_{n_k} = v_{n_k}\}$ により、
$r'.l_i = \textbf{if}\ i \in \{n_1, \ldots, n_k\}\ \textbf{then}\ v_i\ \textbf{else}\ r.l_i$ なる r' を表す。また関数の場合と同様に $r := [\sigma_1, \ldots, \sigma_n] \triangleq ((r := \sigma_1) := \sigma_2) \cdots := \sigma_n$ とする。r_1, r_2 が同じ型のレコードのとき、$r_1 = r_2 := s$ なる s を $r_1 \div r_2$ で表す。 関数及びレコードに対する $:=$ は両オペランドについて strict である。★ .foo の binding power について補足。

2.2 シンタックスの記述

本ドキュメントではシンタックスは非終端記号を <> で囲むスタイルの拡張 BNF で与える。[] で囲むことでオプショナル、{} で囲むことでゼロ回以上の繰り返しを表すとする。また「< と空白」 及び「空白と >」で囲むことでグルーピングを表す。すなわち < *something* > を含んだ BNF 定義はそれらを新たな非終端記号 <*freshname*> で置き換え、定義 <*freshname*> ::= *something* を追加したものと同等である。

BNF 定義において <> で囲まれた記号でそのシンタックスオブジェクトの集合を表す。シンタックスオブジェクトとは与えられたシンタックスを満たす個々の実体のことであり、何らかの方法で数学的なオブジェクトに解釈されているとする。すなわち BNF 定義中の <*foo*> ::= ⋯ は同時に以下のような集合を定義する。スコープは本ドキュメント全体である。

$$foo \triangleq \{w \mid w \text{ は <}foo\text{>より導出される終端記号列 }\}$$

本ドキュメントでは構造をもつデータを記述するために S 式を使う。すなわちシンタックスオブジェクトの汎用的な媒体として S 式を使用するのである。以下の BNF で定義される Sexp が S 式の集合である。この BNF は S 式の具象構文を与えると同時に、上記の規約により S 式の集合 Sexp も定義している。

```
<Sexp>   ::= <Atom> | <List>
<List>   ::= ( {<Sexp>} )
<Atom>   ::= <Symbol> | <String> | <Fixnum> | <Flonum>
<Symbol> ::= シンボル
<String> ::= 文字列
<Fixnum> ::= 整数定数
<Flonum> ::= 浮動小数
```

言葉で書いてある部分の正確な定義は Common Lisp[6] の定義の通りとする。実際にはそれより小さなサブセットでこと足りるが、記述の完全さのために、今はこのように定義しておく。尚、将来的にその小さなサブセットを正式に定義する場合でも、本 S 式は Common Lisp の S 式のサブセットであることを保証する。

LIR においては S 式表現は単にドキュメント記述やデバッグダンプの便宜のために存在するのではなく、LIR という言語の正式な具象構文を与えるために使われる。LIR は後に定義される L モジュールという単位でファイルにセーブし、完全に読み戻すことが出来るように設計されている。よって、S 式を扱える標準的な言語で容易に処理出来るという配慮は意味があるというのがその保証にこだわる理由である。

ここで、Common Lisp の S 式と比較した場合の以下の違いに注意されたい。まず定義から分かるようにドット記法は使用しない。またシンボル `nil` ∈ `Symbol` は空リスト () ∈ `List` と同一視しない。`nil` は単にそういう名前のシンボルに過ぎない。さらに、本ドキュメントで使われるシンボルは全て大文字の英字からなるシンボルに限定されており、大文字の `NIL` というシンボルすら使用しない。よって Common Lisp との上記の食い違いは問題を生じない。以下は S 式の例である。 シンボルは大文字の `TYPEWRITERFONT` で表す。

$$\texttt{SYMBOL} \in \texttt{Symbol} \subset \texttt{Atom} \subset \texttt{Sexp}$$
$$\texttt{"string"} \in \texttt{String} \subset \texttt{Atom} \subset \texttt{Sexp}$$
$$2750 \in \texttt{Fixnum} \subset \texttt{Atom} \subset \texttt{Sexp}$$
$$6.02e23 \in \texttt{Flonum} \subset \texttt{Atom} \subset \texttt{Sexp}$$
$$\texttt{(SYMBOL ("string" 2750) 6.02e23)} \in \texttt{List} \subset \texttt{Sexp}$$

S 式の解釈は次のとおりである。まず `Fixnum`, `Flonum` は Common Lisp の解釈に従い、それぞれ整数 $\boldsymbol{Z}$、実数 $\boldsymbol{R}$ と解釈する。リスト $(x_1 \cdots x_n)$ は $x_i \in X_i$ として X_i の直積の要素 $(x_1, \ldots, x_n)$ と解釈する。文字列は Common Lisp に従った解釈で得られる文字コード (Common Lisp の `char-code` 関数による) のリスト $[c_1, \ldots, c_n] \in [\boldsymbol{Z}]$ と解釈する。シンボルは文字列と同様だがシンボルであることを区別するタグを伴った $(1, [c_1, \ldots, c_n]) \in \boldsymbol{N} \times [\boldsymbol{Z}]$ と解釈する。以下に S 式の解釈について重要な点をまとめる。

1. S 式の等号は上記のように解釈された集合の等号である。従ってシンタックスが等しい二つの S 式は等しい S 式である。
2. S 式の `Fixnum` と `Flonum` は無限の精度を持ち、S 式としてのこれらの要素はそれぞれ数学的な整数及び実数と同一視する。すなわち $\texttt{Fixnum} = \boldsymbol{Z}$, $\texttt{Flonum} = \boldsymbol{R}$。
3. 同じ名前でも文字列とシンボルは区別する。つまり文字列 `"FOO"` とシンボル `FOO` は異なる。

本ドキュメントでは `Sexp` 上の関数として以下を定義する。

$$\begin{aligned}
\# &: \texttt{List} \to \boldsymbol{N} \\
\#(x_1\ x_2\ \cdots\ x_n) &\triangleq n \\
\lambda xn.(x\,!\,n) &: \texttt{List} \to \boldsymbol{N} \to \texttt{Sexp}_\perp \\
x@(x_0\ x_1\ \cdots\ x_n\ \cdots)\,!\,n &\triangleq x_n \quad \textbf{if } n < \#x \\
x\,!\,n &\triangleq \perp \quad \textbf{if } n \geq \#x \\
\lambda ex.(e \in x) &: \texttt{Sexp} \to \texttt{List} \to \boldsymbol{Bool} \\
e \in (x_0\ x_1\ \cdots) &\triangleq \exists x_i\ \ e = x_i \\
\lambda xy.(x +\!\!+\, y) &: \texttt{List} \to \texttt{List} \to \texttt{List} \\
(x_1\ \cdots\ x_n) +\!\!+\, (y_1\ \cdots\ y_m) &\triangleq (x_1\ \cdots\ x_n\ y_1\ \cdots\ y_m)
\end{aligned}$$

$\#\ e$ はリスト e の長さを表す。空リスト () の長さは 0 である。$x\,!\,n$ は $x \in \texttt{List}$ の n 番目の要素、それがなければ $\perp$ を表す。$\in$ は集合論の記号だが、このようにリストにも流用する。また $[\tau]$ 型のリストと同様に $x +\!\!+ y$ で $x, y \in \texttt{List}$ の結合を表す。

シンタックスに関する記述の際、S 式のある部分集合 X の要素からなるリスト全体を正確に表すために以下の表記を導入する。

$$X\ \texttt{List} \stackrel{\triangle}{=} \{(x_0\ x_1\ \cdots) \in \texttt{List} \mid x_i \in X\}$$

尚、S 式の集合としての X List と本ドキュメントで意味記述に使用する型 τ のリスト型 $[\tau]$ は別のものであり、混同しないように注意すること。紛らわしい場合には S 式の List あるいは $[\tau]$ 型のリストというように言葉を区別して使う。意味記述に使う型に S 式の部分集合を使うことは自由であり、$[X]$ という型も許される。しかし X List は S 式によるリストであり、$[X]$ とは別ものである。

S 式のインプリメントの詳細は本ドキュメントの範囲外であるが、基本的なことをまとめておく。S 式は LIR の具象構文にも使われ、実際にファイル中に有限の文字列としてセーブされなければならない。よって数の精度は実際のインプリメントでは何らかの制限が必要である。少なくともコンパイラが対象とするソース言語の精度があれば表現は可能であるが、浮動小数については精度のことを考慮し、ギリギリというのではなく、必要な精度の倍精度表現が理想であろう。

2.3 つまらない定義

本ドキュメントではフォーマルな定義にかならず言葉による説明を添えるが、時として言葉の説明のみで十分曖昧さがなく定義出来、さらに読みやすいフォーマルな定義が与えにくい場合がある。こう判断出来る状況では、無駄な労力を避けてもらうために、そのフォーマルな定義は「**つまらない定義**」である、という文章を添えることにする。

2.4 フォントの使い分けの基本ルール

1. ボールドフォントからなる記号
   一般的概念、意味定義で使われるキーワード。準備のセクションでのみ定義される。(新たに登場した用語はそこでのみボールドで記されるが、それはもちろんここでいうボールドの記号ではない)
   例: ***R***、**if**。

2. 大文字の TYPEWRITERFONT からなる記号
   一切の例外なく、その文字の並びで定まる S 式のシンボルを表す。
   例: `MODULE`。

3. キャピタライズした TypewriterFont からなる記号
   集合、型、型を返す関数。
   例: `Lexp`、`Mem`、`Bit`。

4. 小文字の typewriterfont からなる記号
   関数、定数。
   例: `zb`。

5. 普通の Roman Font からなる記号
   それ自体に数学的な意味は与えない。
   例: Reg、Unspecified。

6. カリグラフ
   シンタックスを引数とし、意味を返す意味関数。
   例: $\mathcal{E}$。

7. イタリック、ギリシャ文字
   記述中で使われる変数を表す。乗算記号は省略しない。したがって ab は a と b の積ではなく、ab という記号であるが、ただし例外としてラムダ記法の束縛変数は 1 文字に限るとする。すなわち $\lambda xy.M = \lambda x.(\lambda y.M)$)。またイタリック体はインフォーマルな文脈においても、誤解の恐れがない場合には <*something*`Modifier>` のように一般的な変数やメタ記号として利用する。

3 概要

ここでは中間言語 LIR を簡単な例によって紹介し、LIR の全体構成について述べる。

3.1 例による LIR の紹介

以下はふたつのソース main.c と sub.c からなる C 言語のプログラムであり、prodv は fold1 を利用して配列 v の要素の積を計算する。

```
main.c :   extern float fold1(float f(float,float), float v[], int n);
           static float v[] = {1, 2.5, 3}
           static int n = sizeof v / sizeof v[0];
           static float fmul(float x, float y){float r=x*y; return r;}
           float prodv(){return fold1(fmul,v,n);}

sub.c :    float fold1(float f(float,float), float v[], int n)
           {
             int i; float r;
             for (r=v[0], i=1; i<n; i++) r = f(r,v[i]);
             return r;
           }
```

これを LIR に変換したものが以下である。LIR は S 式で表現されおり、セミコロンから行末まではコメントと扱う。左の番号は説明のための参照用であり、 LIR の一部ではない。ここに整数、アドレス、浮動小数ともに 32 ビットで、アラインメントの要求は全て 4 バイト境界とする。またコード、データの置かれるセグメント名はそれぞれ text,data とする。まず main.c は以下に変換される。

```
 1  (MODULE "main"
 2    (ALIST
 3      ;; 名前   静的     型      境界 セグメント リンケージ
 4      ("fold1" STATIC UNKNOWN  4  "text"     XREF)
 5      ("v"     STATIC 96       4  "data"     LDEF)
 6      ("n"     STATIC I32      4  "data"     LDEF)
 7      ("fmul"  STATIC UNKNOWN  4  "text"     LDEF)
 8      ("prodv" STATIC UNKNOWN  4  "text"     XDEF))
 9    ;; 関数 fmul の定義
10    (FUNCTION "fmul"
11      (ALIST
12        ;; 名前 フレーム  型  境界  オフセット
13        ("x"    FRAME    F32  4    0)
14        ("y"    FRAME    F32  4    4)
15        ("r"    FRAME    F32  4    8))
16      ;; L 式列
17      (PROLOGUE (12 0) (MEM F32 (FRAME I32 "x")) (MEM F32 (FRAME I32 "y")))
18      (SET F32 (MEM F32 (FRAME I32 "r")) ; r=x*y
19               (MUL F32 (MEM F32 (FRAME I32 "x"))
20                        (MEM F32 (FRAME I32 "y"))))
21      (EPILOGUE (12 0) (MEM F32 (FRAME I32 "r"))))
22    ;; データ v,n の定義
23    (DATA "v" (F32 1.0 2.5 3.0))
24    (DATA "n" (I32 3))
25    ;; 関数 prodv の定義
26    (FUNCTION "prodv"
27      (ALIST
28        ("t1" FRAME F32 4 0)) ; コンパイラが生成した変数
29      (PROLOGUE (4 0))
30      (CALL (STATIC I32 "fold1") ; t1=fold1(fmul,v,n)
31            ((STATIC I32 "fmul") (STATIC I32 "v") (MEM I32 (STATIC I32 "n")))
32            ((MEM F32 (FRAME I32 "t1"))))
33      (EPILOGUE (4 0) (MEM F32 (FRAME I32 "t1")))))
```

これは LIR の L モジュールの例である。L モジュールはモジュール名 (ここではソースファイルに基づいて付けた)、連想リスト、そして関数とデータの定義列から成り立つ。関数定義は関数名、関数の

ローカルな連想リスト、そして L 式列と呼ばれる命令列から成り立つ。データの定義はデータ名とその内容からなり、内容は具体的な値の他に領域を確保する命令、境界制御命令も含む。

連想リストの各エントリはその第一要素の情報を表している。エントリの解釈はその第二要素に依存する。エントリの第二要素が STATIC なら名前は静的なアドレス (最終的にリンカが解決する) を表す。例えば L モジュール "main" の第 6 行は "n" が静的なアドレスを表し、さらにそこには I32 という型 (32 ビット整数) のオブジェクトが置かれており、境界の要求は 4 バイトで data セグメントに割り当てられ、リンケージはローカル定義 LDEF すなわちモジュール外から参照出来ないということを意味する。リンケージには他に外部に対して定義することを意味する XDEF と、外部参照を意味する XREF がある。

第 6 行の v の型は単に 96 とあるが、これは 96(=32*3) ビットのオブジェクトであることを意味する型である。他に LIR は浮動小数用の型 (例えば第 13 行にある F32) もある。 LIR の型システムはこのようにハードが直接認識出来るものとなっている。これを L タイプという。コンパイル時にその実際の大きさが分からない場合は UNKNOWN としているが、これは L タイプではない。

第 10 行は簡単な関数定義の例である。L 式列の最初と最後の命令 PROLOGUE、EPILOGUE は関数としてのインタフェースを定義している。これは実際のハードでは通常存在しない命令であるが、LIR ではこのようなインタフェースを付加することにより、コード最適化において幾つかの定理を提供している。一般に LIR では関数は多値関数であり、PROLOGUE の第三要素以降が値を受けとる変数のならび、EPILOGUE の第三要素以降が返される多値を表す式のならびである。

PROLOGUE、EPILOGUE の L 式の第二要素 $(w_f\ w_r)$ はフレームのサイズを表す。最初の w_f はフレームのサイズだが、次の w_r はレジスタフレームのサイズである。これらについては後に説明する。

関数定義中の連想リストのエントリ (例えば第 13 行) は、その名前がフレーム中のオフセットを表すものであることを示す FRAME を第二要素として持つ。このエントリでは型、境界、フレーム中のオフセットが要素である。

第 18 行が典型的な L 式である。対応する C 言語のコードと比較されたい。まずメモリーの参照 (読み書き) は明示的に (MEM 型 アドレス) という L 式で表されている点が C 言語と異なる。型はそのメモリー参照が表すオブジェクトの型である。そして、たとえば第 13 行で定義されている "x" は、フレームに割り当てられた変数のアドレスを表す L 式 (フレーム式) により (FRAME I32 "x") と表される。ここで I32 はこの L 式の型すなわちアドレスの型である。

フレームポインタは様に現れていない。これによりコード最適化フェーズでの処理が容易となる。しかし LIR にはフレームポインタも存在し、フレーム式の意味定義ではそれが参照されている。詳細はここでは述べない。

第 30 行が LIR の関数呼びだし命令であり、多値を返すために (CALL 関数アドレス (引数...) (結果を入れる変数...) というシンタックスとなっている。関数呼びだしは他の命令の部分にはなれない。よって t1 という一時変数が導入されている。

第 31 行の (MEM I32 (STATIC I32 "n")) はグローバル変数の参照例であり、第 7 行で宣言された "n" がスタティックなアドレスを表す L 式により (STATIC I32 "n") と表されている。

この例で欠けている重要なものにレジスタがある。LIR のレジスタは (REG 型 "regname") というシンタックスで表され、"regname" は連想リストにおいては ("regname" REG 型 オフセット) というエントリで表される。レジスタの名前が表すものはそれが保持するオブジェクトであり、レジスタのアドレスではない。そもそもレジスタのアドレスというものは通常存在しないが、LIR ではレジスタメモリというものを仮定し、各レジスタはそのメモリ中に置かれるとして扱う。上のレジスタのエントリにおけるオフセットとはそのレジスタのアドレスを決定するのに使われる。先に触れたレジスタフレームのサイズ w_r もまたそのアドレスの決定に使われる。詳細はここでは説明しない。

本来、関数の引数のいくつかは特定の方法によって特定のレジスタ経由で渡される。それらのレジスタをどのタイミングで明示化するかが問題であるが、我々は引数を明示した PROLOGUE、EPILOGUE の導入により、その処理を最終的なレジスタアロケーションにおいて行なう。これによりコード最適化フェー

ズにおいては具体的なハードレジスタをケアする必要がなくなるのである。LIR は最終的なコード生成まで使われる表現であり、我々はコード生成も含めて全てのパスを LIR のプログラム変換と捉える。例えば先の fmul の引数がレジスタ渡しされ、それらが caller save レジスタであったならば、それは次のようにプログラム変換されることになる。ここに、モジュールの連想リストは制約としてプログラム全体を構成する他のモジュールの連想リストと整合していなければならない (整合の詳しい定義は後にする)。これによりモジュール連想リストに存在するレジスタは実際のハードレジスタを表すと見なす。LIR ではこのようなプログラム変換としてレジスタアロケーションをとらえる。

```
(MODULE "main"
  (ALIST
    .......
    ("F0" REG F32 0) ; ハードレジスタ F0
    ("F1" REG F32 4) ; ハードレジスタ F1
    ......

  (FUNCTION "fmul"
    (ALIST )
    (PROLOGUE (0 0) (REG F32 "F0") (REG F32 "F1"))
    (SET F32 (REG F32 "F0") (MUL F32 (REG F32 "F0") (REG F32 "F1")))
    (EPILOGUE (0 0) (REG F32 "F0")))
  .....
```

次に sub.c に対応する L モジュール "sub" を説明する。

```
 1  (MODULE "sub"
 2    (ALIST
 3      ("fold1" STATIC UNKNOWN 4 "text" XDEF))
 4    (FUNCTION "fold1"
 5      (ALIST
 6        ("f" FRAME I32 4 0)
 7        ("v" FRAME I32 4 4)
 8        ("n" FRAME I32 4 8)
 9        ("i" FRAME I32 4 12)
10        ("r" FRAME F32 4 16))
11      (PROLOGUE (20 0) (MEM I32 (FRAME I32 "f"))
12                       (MEM I32 (FRAME I32 "v"))
13                       (MEM I32 (FRAME I32 "n")))
14      (SET F32 (MEM F32 (FRAME I32 "r")) ; r=v[0]
15               (MEM F32 (MEM I32 (FRAME I32 "v"))))
16      (SET I32 (MEM I32 (FRAME I32 "i")) ; i=1
17               (INTCONST I32 1))
18      (DEFLABEL "L1")
19      (JUMPC (TSTLT I32 (MEM I32 (FRAME I32 "i")) ; if (i<n) goto L2; else goto L3
20                        (MEM I32 (FRAME I32 "n"))) (LABEL I32 "L2") (LABEL I32 "L3"))
21      (DEFLABEL "L2")
22      (CALL (MEM I32 (FRAME I32 "f")) ; r=f(r,v[i])
23            ((MEM F32 (FRAME I32 "r"))
24             (MEM F32 (ADD I32 (MEM I32 (FRAME I32 "v")) ; v[i]
25                               (MUL I32 (MEM I32 (FRAME I32 "i"))
26                                        (INTCONST I32 4)))))
27            ((MEM F32 (FRAME I32 "r"))))
28      (SET I32 (MEM I32 (FRAME I32 "i")) ; i++
29               (ADD I32 (MEM I32 (FRAME I32 "i"))
30                        (INTCONST I32 1)))
31      (JUMP (LABEL I32 "L1"))
32      (DEFLABEL "L3")
33      (EPILOGUE (20 0) (MEM F32 (FRAME I32 "r")))))
```

この関数定義は制御構造を含む例である。いわゆる構造化プログラミング向けの高級な制御構造は提供せず、代わりにいくつかの条件分岐命令を提供する。(DEFLABEL "ラベル") によりラベルを定義し、参照は (LABEL 型 "ラベル") で行なう。この型はコードのアドレスを表す整数の適当な型である。JUMP は無条件分岐、JUMPC は条件分岐である。他に JUMPN という多方向分岐命令もある。

ラベルのスコープは関数内であり、関数の外のラベルへ分岐することは出来ない。むろん、ラベル名は最終的に生成されるアセンブラ言語にそのまま使われるなら、アセンブラでは関数ローカルなラベルというものは存在しないので、適当なリネーム処理が必要とされる。しかし LIR の言語仕様上はあくまでラベルのスコープは関数内と定める。

第 22 行は関数の間接呼びだしの例にもなっている。少なくとも C 言語を表現する必要上、間接呼びだしは必要であり、CALL 式の第一引数はしたがって関数のアドレスを表す任意の式が書けるようになっている。しかし LIR のモデルではこの計算されたアドレスは必ず LIR で書かれた関数であると考える。

3.2 LIR の構成

以上の例で説明された LIR の構成要素を、正式な構文名と意味関数を添えて説明する。これらを定義していくのが残りのセクションの目的である。LIR は以下の 8 つの構成要素からなる。構文名とはその構成要素のシンタックスオブジェクトの集合を表す記号、意味関数とはその構成要素の意味を与える関数である。

構成要素	意味	構文名	意味関数
L プログラム	ひとつのプログラム	Lprog	$\mathcal{P}$
L モジュール	プログラムを構成するモジュール	Lmod	$\mathcal{M}$
L 連想リスト	テーブル	Lalist	$\mathcal{A}$
L データ	プログラムのデータ部分	Ldata	$\mathcal{D}$
L 関数	プログラムの関数部分	Lfunc	$\mathcal{F}$
L 式列	命令列	Lseq	$\mathcal{S}$
L 式	ひとつの命令	Lexp	$\mathcal{E}$
L タイプ	型	Ltype	$\mathcal{T}$

各構成要素を下から説明する。**L タイプ**は LIR の型であり、現在のハードが直接扱える型からなる。**L 式**はひとつの命令を表し、**L 式列**はその命令の並びを表す。先の例においては PROLOGUE で始まり、EPILOGUE で終る例のみであったが、L 式列 (Lseq) は単に L 式のリストとして定義する。**L 関数**は関数ローカルな連想リストと、多値関数のインタフェースを伴った L 式列で構成される。**L データ**はデータセグメントを表現する。**L 連想リスト**は変数、関数、レジスタを表す名前にその意味する値を対応させる構造であり、先の例での使用が全ての場合である。すなわちそれはモジュールの最初か関数定義の最初に現れる。そして連想リスト、関数、データの定義を伴った**L モジュール**がひとつのコンパイル単位となる。ひとつの L モジュールは一般に外部定義や外部参照を含む。それらが解決するように関連しあった L モジュールの集合が **L プログラム**であり、LIR のトップレベルの構造である。

★ Portable MAD Function SET

4 LIR のシンタックス

ここでは LIR のシンタックスを二つに分けて定義する。前半「L 式のシンタックス」では L 式及びその部分的なシンタックスを定義する。後半「L プログラムのシンタックス」では L プログラムまでの上位の構造を定義する。

4.1 シンタックスの定義

LIR のシンタックスは S 式の部分集合となるように定義し、またキーワードに相当するものを大文字のシンボルで表す。レジスタ名などの名前は全て文字列で表す。左端の番号は参照用であり、構文の一部ではない。この BNF 定義では以下の非終端記号を外部参照している。

```
<Sexp>   : S 式
<String> : S 式の文字列
<Fixnum> : S 式の整数定数
<Flonum> : S 式の浮動小数
```

概要で説明したように、セマンティックスを与える対象となる非終端記号の名前は L というプリフィックスを付けるが、その他の記号については L プリフィックスは付けない。

S 式の慣例に従い、セミコロンから行末まではコメントと扱う。また、Common Lisp に従い #| から |# までをコメントとして扱う。LIR のシンタックスは S 式に基づいているため、これを具象構文と考えてファイルなどにセーブしたり、読み戻したりするためのシンタックスに流用出来る。LIR のリーダ (構文解析プログラム) はこのコメントの処理をサポートしなければならない。

L 式のシンタックス

```
 1  <Lexp>               ::= <TypedExp> | <UnTypedExp>
 2
 3  <TypedExp>           ::= <AtomicTypedExp> | <NonAtomicTypedExp>
 4  <AtomicTypedExp>     ::= <ConstExp> | <AddrExp> | <RegExp>
 5  <NonAtomicTypedExp> ::= <PureExp> | <MemExp> | <SetExp>
 6
 7  <UnTypedExp>         ::= <JumpExp> | <DefLabelExp> | <CallExp> | <InterfaceExp>
 8                           | <SpecialExp>
 9
10  <Ltype>              ::= <Itype> | <Ftype> | <Fixnum>
11  <Itype>              ::= I8 | I16 | I32 | I64 | I128
12  <Ftype>              ::= F32 | F64 | F128
13
14  <ConstExp>           ::= <IntConstExp> | <FloatConstExp>
15  <IntConstExp>        ::= (INTCONST   <Ltype> <Fixnum>)
16  <FloatConstExp>      ::= (FLOATCONST <Ltype> <Flonum>)
17
18  <AddrExp>            ::= <StaticExp> | <FrameExp> | <LabelExp>
19  <StaticExp>          ::= (STATIC <Ltype> <String>)
20  <FrameExp>           ::= (FRAME  <Ltype> <String>)
21  <LabelExp>           ::= (LABEL  <Ltype> <String>)
22
23  <RegExp>             ::= <SimpleRegExp> | <SubRegExp>
24  <SimpleRegExp>       ::= (REG <Ltype> <String>)
25  <SubRegExp>          ::= (SUBREG <Ltype> <SimpleRegExp> <Fixnum> [& <Modifier>])
26
27  <PureExp>            ::= (<PureOps> <Ltype> <TypedExp> {<TypedExp>} [& <Modifier>])
28  <PureOps>            ::= <ArithOps> | <ConvOps> | <BitOps> | <ShiftOps> | <TstOps>
29                           | ASMCONST | PURE
30  <ArithOps>           ::= NEG | ADD | SUB | MUL | DIVS | DIVU | MODS | MODU
31  <ConvOps>            ::= CONVSX | CONVZX | CONVIT | CONVFX
32                           | CONVFT | CONVFI | CONVSF | CONVUF
33  <BitOps>             ::= BAND | BOR | BXOR | BNOT
34  <ShiftOps>           ::= LSHS | LSHU | RSHS | RSHU
35  <TstOps>             ::= TSTEQ | TSTNE | TSTLTS | TSTLES | TSTGTS | TSTGES
36                           | TSTLTU | TSTLEU | TSTGTU | TSTGEU
37
38  <MemExp>             ::= (MEM <Ltype> <TypedExp> [& <Modifier>])
39
40  <SetExp>             ::= (SET <Ltype> < <MemExp> | <RegExp> > <TypedExp>)
41
42  <JumpExp>            ::= (JUMP <LabelExp>)
43                           | (JUMPC <TypedExp> <LabelExp> <LabelExp>)
44                           | (JUMPN <TypedExp> ( { (<Fixnum> <LabelExp>) } ) <LabelExp>)
45
46  <DefLabelExp>        ::= (DEFLABEL <String>)
47
48  <CallExp>            ::= (CALL <TypedExp> ( {<TypedExp>} ) ( {<InterfaceVar>} ))
49  <InterfaceVar>       ::= <RegExp> | (MEM <Ltype> <FrameExp>)
50  <InterfaceExp>       ::= <PrologueExp> | <EpilogueExp>
51  <PrologueExp>        ::= (PROLOGUE (<Fixnum> <Fixnum>) {<InterfaceVar>})
52  <EpilogueExp>        ::= (EPILOGUE (<Fixnum> <Fixnum>) {<TypedExp>})
53
54  <SpecialExp>         ::= <ParallelExp> | <UseExp> | <ClobberExp>
55  <ParallelExp>        ::= (PARALLEL {<SetExp> | <CallExp> | <UseExp> | <ClobberExp>})
56  <UseExp>             ::= (USE <RegExp>)
57  <ClobberExp>         ::= (CLOBBER < <RegExp> | <MemExp> >)
58
59  <Modifier>           ::= <ShiftModifier> | <SubregModifier> | <MemModifier>
60  <ShiftModifier>      ::= (< S | U > <Fixnum> < D | U >)
61  <SubregModifier>     ::= S | N
62  <MemModifier>        ::= N | V
```

L プログラムのシンタックス

```
 1  <Lprog>       ::= ( {<Lmod>} )
 2
 3  <Lmod>        ::= (MODULE <String> <GlobalAlist> {<Ldata> | <Lfunc>})
 4
 5  <GlobalAlist> ::= (ALIST {<GlobalEnt>} )
 6  <LocalAlist>  ::= (ALIST {<LocalEnt>} )
 7  <Lalist>      ::= <GlobalAlist> | <LocalAlist>
 8
 9  <GlobalEnt>   ::= (<String> < <RegEnt> | <StaticEnt> >)
10  <LocalEnt>    ::= (<String> < <RegEnt> | <FrameEnt>  >)
11  <Ent>         ::= <GlobalEnt> | <LocalEnt>
12
13  <RegEnt>      ::= REG    <Ltype> <Offset>
14  <FrameEnt>    ::= FRAME  <Ltype> <Align> <Offset>
15  <StaticEnt>   ::= STATIC < <Ltype> | UNKNOWN > <Align> <Segment> <Linkage>
16  <EntClass>    ::= REG | FRAME | STATIC
17  <Align>       ::= <Fixnum>
18  <Offset>      ::= <Fixnum>
19  <Segment>     ::= <String>
20  <Linkage>     ::= LDEF | XDEF | XREF
21
22  <Ldata>       ::= (DATA <String> {<DataSeq> | <ZeroSeq> | <SpaceSeq>})
23  <DataSeq>     ::= (<Ltype> {<Fixnum> | <Flonum> | <Lexp>})
24  <ZeroSeq>     ::= (ZEROS <Fixnum>)
25  <SpaceSeq>    ::= (SPACE <Fixnum>)
26
27  <Lfunc>       ::= (FUNCTION <String> <LocalAlist> <Lseq>)
28  <Lseq>        ::= ( {<Lexp>} )
```

4.2 シンタックスに関する用語、記号の定義

L モジュール (MODULE $name$ $alist$ $\cdots$) $\in$ Lmod の $name$ を**モジュール名**、$alist$ を **グローバル連想リスト**と呼ぶ。また、L 関数 (FUNCTION $name$ $alist$ seq) $\in$ Lfunc の $name$ を **L 関数名** と呼び、$alist, seq$ をそれぞれ L 関数の **ローカル連想リスト**、L 式列という。これらについて以下の記法を定義する。

$$
\begin{aligned}
\lambda m.(m.\texttt{name}) &: \texttt{Lmod} \to \texttt{String} \\
(\texttt{MODULE}\ name\ \cdots).\texttt{name} &\triangleq name \\
\lambda m.(m.\texttt{alist}) &: \texttt{Lmod} \to \texttt{GlobalAlist} \\
(\texttt{MODULE}\ n\ alist\ \cdots).\texttt{alist} &\triangleq alist \\
\lambda f.(f.\texttt{name}) &: \texttt{Lfunc} \to \texttt{String} \\
(\texttt{FUNCTION}\ name\ \cdots).\texttt{name} &\triangleq name \\
\lambda f.(f.\texttt{alist}) &: \texttt{Lfunc} \to \texttt{LocalAlist} \\
(\texttt{FUNCTION}\ n\ alist\ \cdots).\texttt{alist} &\triangleq alist \\
\lambda f.(f.\texttt{seq}) &: \texttt{Lfunc} \to Lseq \\
(\texttt{FUNCTION}\ n\ a\ seq\ \cdots).\texttt{seq} &\triangleq seq
\end{aligned}
$$

L 連想リスト (ALIST e_1 e_2 $\cdots$) $\in$ Lalist の e_i を連想リストの**エントリ**という。各エントリ e_i の第一要素は名前 (String) 第二要素はシンボルである。この名前を**エントリ名**、シンボルをエントリ e_i の**エントリクラス**という。

$$
\begin{aligned}
\lambda e.(e.\texttt{name}) &: \texttt{Ent} \to \texttt{String} \\
(name\ class\ \cdots).\texttt{name} &\triangleq name \\
\lambda e.(e.\texttt{class}) &: \texttt{Ent} \to \texttt{EntClass} \\
(name\ class\ \cdots).\texttt{class} &\triangleq class
\end{aligned}
$$

各エントリのエントリクラス以下のシンタックスはそのエントリクラスに依存する。以下の関係に注意 (直和ではない)。

$$\texttt{Lalist} = \texttt{GlobalAlist} \cup \texttt{LocalAlist}$$

集合 FooExp の要素を Foo 式と呼ぶ。例えば「r は Reg 式である」という表現は「$r \in$ RegExp」と同じであり、「Reg 式 r について ...」は「$r \in$ RegExp について ...」と同じである。

式 $e \in$ Lexp はリストの形をしている。その第一要素は機能を定めるシンボルであり、**L 式のコード**という。L 式 e のコードを e.code で表す。コードが FOO である式を FOO 式ともいう。Typed 式の場合は第二要素にその式の型を表す L タイプがくる。それを **L 式の型**といい、e.type で表す。

$$
\begin{aligned}
\lambda e.(e.\texttt{code}) &: \texttt{Lexp} \to \texttt{Symbol} \\
(code\ \cdots).\texttt{code} &\triangleq code \\
\lambda e.(e.\texttt{type}) &: \texttt{TypedExp} \to \texttt{Ltype} \\
(code\ type\ \cdots).\texttt{type} &\triangleq type
\end{aligned}
$$

L 式 e のコードと型 (もしあれば) 以後に現れている L 式を出現順にならべたものを **L 式の引数リスト** といい、 e.args で表す。正確な定義は以下のとおりである。

$$
\begin{aligned}
\lambda e.(e.\texttt{args}) &: \texttt{Lexp} \to \texttt{Lexp List} \\
e.\texttt{args} &\triangleq () && \textbf{if } e \in \texttt{AtomicTypedExp} \\
e@(op\ type\ x_1\ \cdots\ x_n\ [\&\ m]).\texttt{args} &\triangleq (x_1\ \cdots\ x_n) && \textbf{if } e \in \texttt{NonAtomicTypedExp} \\
(\texttt{JUMP}\ l_1).\texttt{args} &\triangleq (l_1) \\
(\texttt{JUMPC}\ x\ l_1\ l_2).\texttt{args} &\triangleq (x\ l_1\ l_2) \\
(\texttt{JUMPN}\ x\ ((c_1\ l_1)\cdots(c_n\ l_n))\ l_0).\texttt{args} &\triangleq (x\ l_1\ \cdots\ l_n\ l_0) \\
(\texttt{DEFLABEL}\ s).\texttt{args} &\triangleq () \\
(\texttt{CALL}\ x_0\ (x_1\ \cdots\ x_n)\ (y_1\ \cdots\ y_m)).\texttt{args} &\triangleq (x_0\ x_1\ \cdots\ x_n\ y_1\ \cdots\ y_m) \\
(\texttt{PROLOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n).\texttt{args} &\triangleq (x_1\ \cdots\ x_n) \\
(\texttt{EPILOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n).\texttt{args} &\triangleq (x_1\ \cdots\ x_n) \\
(\texttt{PARALLEL}\ x_1\ \cdots\ x_n).\texttt{args} &\triangleq (x_1\ \cdots\ x_n) \\
(\texttt{USE}\ r).\texttt{args} &\triangleq (r) \\
(\texttt{CLOBBER}\ x).\texttt{args} &\triangleq (x)
\end{aligned}
$$

L 式 e について $(a_1\ \cdots\ a_n) \triangleq e.\texttt{args}$ とするとき、a_i を **直接の部分 L 式** あるいは第 i **引数 L 式**という。上記定義の最初の二行が Typed 式に関するものである。AtomicTyped 式の引数リストは空である。それ以外の Typed 式は型に続く L 式でモディファイア (後述) を除く部分が引数リストとなる。 これ以外の定義は UnTyped 式に関するものである。

x が e の直接の部分式である、という関係を $x \in_{\mathbf{a}} e$ で表す。またその推移的閉包を**部分 L 式** といい、 $\in^+_{\mathbf{a}}$ で表す。同様にその反射的推移的閉包を**広義の部分 L 式**といい、$\in^*_{\mathbf{a}}$ で表す。

$$
\begin{aligned}
\lambda xe.(x \in_{\mathbf{a}} e),\ \lambda xe.(x \in^+_{\mathbf{a}} e),\ \lambda xe.(x \in^*_{\mathbf{a}} e) &: \texttt{Lexp} \to \texttt{Lexp} \to \boldsymbol{Bool} \\
x \in_{\mathbf{a}} e &\triangleq x \in e.\texttt{args} && \text{(直接の部分式、引数)} \\
x \in^+_{\mathbf{a}} e &\triangleq x \in_{\mathbf{a}} e \vee \exists x_i \in e.\texttt{args}\ x \in^+_{\mathbf{a}} x_i && \text{(部分式)} \\
x \in^*_{\mathbf{a}} e &\triangleq x = e \vee x \in^+_{\mathbf{a}} e && \text{(広義の部分式)}
\end{aligned}
$$

L 式列 $es \triangleq (e_0\ e_1\ \cdots)$ において e_i を es の**トップレベル L 式**という。 $es\,!\,i = e$ のとき、es の**ロケーション** i の L 式が e であるという。

4.3 シンタックスレベルでの制約

ここでは BNF で表現出来ていないシンタックス上の制約を定義する。セマンティックスが定義可能なシンタックスオブジェクト X を正しい X という。例えば正しい L 式、正しい L プログラム、などと表現する。以下はその判定関数である。

$$
\begin{aligned}
\texttt{validprog} &: \texttt{Lprog} \to \boldsymbol{Bool} \\
\texttt{validmod} &: \texttt{Lmod} \to \boldsymbol{Bool} \\
\texttt{validalist} &: \texttt{Lalist} \to \boldsymbol{Bool} \\
\texttt{validdata} &: \texttt{Ldata} \to \boldsymbol{Bool} \\
\texttt{validfunc} &: \texttt{Lfunc} \to \boldsymbol{Bool} \\
\texttt{validseq} &: \texttt{Lseq} \to \boldsymbol{Bool} \\
\texttt{validexp} &: \texttt{Lexp} \to \boldsymbol{Bool} \\
\texttt{validltype} &: \texttt{Ltype} \to \boldsymbol{Bool}
\end{aligned}
$$

以下は validltype, validexp, validseq の定義である。補助関数として validexp_set, validexp_sem を使っており、前者は Set 式の出現についての制約、後者は各 L 式に依存する制約を表す。

$$
\begin{aligned}
\texttt{validltype}\,t &\triangleq t \in \boldsymbol{Z} \rightarrow 0 < t \wedge t \equiv 0 \,(\mathrm{mod}\, 8)\\
\texttt{validexp}\,e &\triangleq \texttt{validexp\_set}\,e \wedge \forall x \in^{*}_{\mathbf{a}} e\ \texttt{validexp\_sem}\,x\\
\texttt{validexp\_set}\,e &\triangleq \mathbf{if}\ e \in \texttt{ParallelExp}\ \mathbf{then}\ \forall x \in_{\mathbf{a}} e\ \ \texttt{validexp}\,x\\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \ \forall x \in^{+}_{\mathbf{a}} e\ \ x \notin \texttt{SetExp}\\
\texttt{validexp\_sem}\,e &\triangleq \text{後に意味記述の所で定義される}\\
\texttt{validseq}\,s &\triangleq \wedge\ \ \forall e \in s\ \ \texttt{validexp}\,e\\
&\qquad \wedge\ \ \forall e \in s\ \ e \in \texttt{UnTypedExp} + \texttt{SetExp}\\
&\qquad \wedge\ \ \texttt{mklabelenv}\,s \neq \texttt{DUP}\\
&\qquad \wedge\ \ \forall e \in s\ \ (\exists x \in^{+}_{\mathbf{a}} e\ \ x \in \texttt{LabelExp}) \rightarrow e \in \texttt{JumpExp}\\
&\qquad \wedge\ \ \forall e \in s\ \ (\forall x \in^{+}_{\mathbf{a}} e\ \ x \in \texttt{LabelExp} \rightarrow (\texttt{mklabelenv}\,s)\,(x\,!\,2) \neq \bot)
\end{aligned}
$$

ここに mklabelenv は L 式列を引数とし、ラベル文字列にそれが定義 (DEFLABEL) されているロケーションを対応させる部分関数 (後に定義される L 環境の LabelEnv) を返す関数である。ただし二重定義が存在したらシンボル DUP を返す。この関数の「つまらない定義」が後の L 環境の定義の所 (p.35) で与えられる。

すなわち正しい L タイプは、それが Fixnum であれば正の値でなければならない。正しい L 式は Parallel 式を除き、Set 式を部分式として含んではならない。また正しい L 式列は正しい L 式のリストで、トップレベル式が UnTyped 式と Set 式のみからなるものであり、ラベル式は Jump 式のオペランド以外には現れず、Label 式に対応するラベルは一意的に DefLabel 式で定義されていなければならない。validexp_sem は後に L 式のセマンティックスの所で定義される。

以下は validalist の定義であり、エントリ名が重ならないという条件である。

$$
\texttt{validalist}[\![(\texttt{ALIST}\,e_1\ \cdots\ e_n)]\!] \triangleq \forall e_i\ \forall e_j\ \ e_i\,!\,0 = e_j\,!\,0 \rightarrow i = j
$$

★ validfunc, validdata, validmod, validprog の定義がまだ。

> BNF で定義されたシンタックスオブジェクトの集合は以後ここで述べられた制約を全て満たしているとする。

- 議論

Q. L タイプは拡張出来ないのか。

A. 何をもって「拡張出来た」と言えるのかが明確ではない。理想的には単にある記述ファイルに I256 などと追加するだけで全てうまく行くということになるだろう。しかしそういうことは今予定されていない。もちろんインプリメント上はソースをいじって拡張しやすいように設計されているはずであり、ソースは公開されているのだから、そういう意味では拡張可能である。しかし、どの程度拡張しやすければ拡張可能と主張できるのか COINS では定義がない。

Q. Set 式がなぜ TypedExp に分類されるのか。制約より Set 式の値は捨てられるだけだが。

A. C 言語にあるような副作用を引数にもつ式を LIR で自然に表現したいという下心があるからである。実はセマンティックスの記述自体は Set に関する制約を外してもうまく行くように定義している。つまり部分式を左から右へと評価し、その順で副作用が累積されるという形で定義されている。この下心を将来的に正式なドキュメントとするか、あるいは廃止するかは未定である。
これとは別の言い分けとして、 Set 式に型があるとマシン記述のパターンが多少分かりやすいのではないか (引数の型を見るよりも)、ということがある。

4.4 モディファイアについて

L 式の意味を「微調整」する必要から、幾つかの L 式はそのリストの最後に `& <Modifier>` なるシンタックスで **L 式のモディファイア**が指定出来るようになっている。この `<Modifier>` は正式な引数 (L 式) ではなく微調整の情報を表現する S 式である。 L 式 e のモディファイアを e.`mod` で表す。

$$\begin{aligned} \lambda e.(e.\texttt{mod}) &: \texttt{Lexp} \to \texttt{Sexp} \\ (\cdots \ \& \ m).\texttt{mod} &\stackrel{\triangle}{=} m \\ e.\texttt{mod} &\stackrel{\triangle}{=} \texttt{NONE} \quad \textbf{if } \& \notin e \end{aligned}$$

ごく限られた L 式のみが Modifier をもつ。構文上は BNF の繁雑さを避けるためにそのような L 式を厳密には区別していない。`<`*something*`Modifier>` の *something* が、その Modifier の対象となる L 式を特定する名前となっている。下は現在定義されているモディファイアとその対象及びその直観的な意味である。正確な意味は後にセイマンティックスの所で与える。

Modifier 構文	対象となる L 式のコード	直観的な意味
`<ShiftModifier>`	`LSHS, LSHU, RSHS, RSHU`	シフト演算の細かい違い
`<SubregModifier>`	`SUBREG`	部分レジスタへの代入の挙動の違い
`<MemModifier>`	`MEM`	メモリ参照の最適化の制限

- 議論

  Q. これは汚いが、どうしても必要なのだろうか。

  A. 汚いかという問に対する答えはもちろん Yes である。どうしても必要かという問に対する答えは No であり、他の手段も考えられると思う。これは幾つかの案を考慮した上での現在の我々の結論である。Modifier の意味を理解した上で、もしより良いアイディアが見つかったら教えていただきたい。

5 LIR のセマンティックス

本セクションでは以下の意味関数の定義を与える。意味関数は表示的意味論に従い数学的な関数として与える。

$\mathcal{P}$: Lprog → L プログラムの意味
$\mathcal{M}$: Lmod → L モジュールの意味
$\mathcal{A}$: Lalist → L 連想リストの意味
$\mathcal{D}$: Ldata → L データの意味
$\mathcal{F}$: Lfunc → L 関数の多値関数モデルによる意味
$\mathcal{S}$: Lseq → L 式列の 1 ステップ実行の意味
$\mathcal{E}$: Lexp → L 式の意味
$\mathcal{T}$: Ltype → L タイプの意味

L 式列はいわばコードセグメントに置かれたマシン命令列を意味している。各 L 式 e_i は順時実行されることでメモリに対して読み書きを行なう。メモリは L 式列の一部でなない。したがって別に提供する必要がある。これを **L メモリ**と呼ぶ。また一部の L 式はレジスタやアドレスを表しているので、それらの対応も与える必要がある。これを **L 環境**と呼ぶ。

ここではまず L メモリと L 環境の直観的意味を説明し、それを定式化する。そしての実行環境における L 式の動作の意味を与えることにより意味関数を定義する。

5.1 L メモリと L 環境の直観的意味

L メモリは以下のものからなる。それぞれに関連する L 式の直観的な意味も説明する。ここで $A \in$ Itype はアドレスを表す型である。

1. プログラムカウンター pc。
   L 関数の L 式列 $(e_0\ e_1\ \cdots)$ で次に実行すべき命令のロケーションを示す。この pc は LIR のレジスタでは表現できない。pc $= i$ のとき e_i の実行後、pc は通常 1 増えるが、Jump 命令の場合は Label 式の表すロケーションが設定される。

2. プログラムメモリ pm。
   L 関数が置かれているメモリ。命令がメモリにどのように配置されるかはマシンに依存しすぎるし、その表現を必要とする最適化は特殊なものに限られる。よって我々はプログラムメモリが保持する対象を個々の命令の具体的ビットパターンではなく、シンタックスオブジェクトとしての L 関数とする。本来プログラムメモリはセマンティックスの世界に属するものであるが、中間言語ではプログラム変換が重要であるから、プログラムメモリが保持するものはシンタックスオブジェクトとしている。このメモリにアクセス出来るのは唯一 Call 式のみであり、それも L 関数のセマンティックスを通じてのアクセスに制限する。このようにモデル化しても、上でいう特殊な最適化をこの枠組で行なうことは可能である。詳細はプラグマティックスで説明する。

3. レジスタメモリ rm。
   汎用レジスタ群が置かれているメモリ。意味定義の都合によりレジスタは次のデータメモリとはアドレス空間の異なるレジスタメモリに置かれているとする。よってレジスタのアドレスが存在する。しかしそれはデータメモリのアドレス空間とは異なり、データメモリにアクセスする命令によってレジスタをアクセスすることは出来ない。

   Reg 式 (REG t s) は $s \in$ String で指定される型 t のレジスタが格納している値を表す。また Set 式 (SET t (REG t s) v) は型 t の Typed 式 v の値をそのレジスタの新しい値とするように書き込みを行なう。特に "@FP"という文字列で表されるレジスタを**フレームポインタ**という。

4. データメモリ dm。
   データの置かれている読み書き可能なメモリ。型 A の任意の Typed 式は dm のアドレスを表すと解釈出来る。Static 式 (STATIC A s) は $s \in$ String で指定される dm のアドレスを表す。Frame 式 (FRAME A s) は $s \in$ String で示されるオフセット値を現在のフレームポインタの値に加えて生成される dm のアドレスを表す。このとき s を **フレーム変数**と呼ぶ。

   Mem 式 (MEM t a) は dm のアドレス a にある型 t の **オブジェクト**を意味し、それが格納している値を表す。また Set 式 (SET t (MEM t a) v) は型 t の Typed 式 v の値をそのオブジェクトの新しい値とするように書き込みを行なう。

5. トレース tr。
   メモリへの参照により IO を行なうマシンでは関数の意味として特定のメモリへの読み書きを時系列として記録したリストを考慮する必要がある。そのリストを保持するための変数である。このような変数は実際のハードウエアには存在しない。これはあくまで意味記述の便宜上の変数である。読み書きがトレースへ記録されるオブジェクトを **volatile オブジェクト** という。volatile オブジェクトの意味定義には次のランダムステートも関連する。

6. ランダムステート rs。
   意味記述で使われる乱数発生関数 random の状態 (実数) を保持する変数。ここを参照するのはこの random 関数のみである。乱数は意味記述において、ある特定の値を予測することで可能となるような (あやまった) 最適化を禁止するために使われる。例えば volatile 変数 x について x-x を 0 とするのは誤りである。我々は volatile 変数に、その値は常に乱数であるという意味づけをすることで、この問題に対処している。

L 環境は以下のものからなる。

1. Reg 式の対応 reg。
   関数中の全ての Reg 式 (REG t s) の $s \in$ String に rm 中の適当なアドレス reg$s$ を対応させる。これにより (REG t s) は rm 中のそのアドレスに置かれた型 t のレジスタを表すことになる。

2. Static 式の対応 sta。
   関数中の全ての Static 式 (STATIC A s) の $s \in$ String に dm 中の適当なアドレス sta$s$ を対応させる。これにより (STATIC A s) は dm 中のそのアドレスを表すことになる。

3. Frame 式の対応 fra。
   関数中の全ての Frame 式 (FRAME A s) の $s \in$ String にフレームポインタからの適当なオフセット値 fra$s$ を対応させる。これにより (FRAME A s) は現在のフレームポインタ値とそのオフセット値の和で作られる dm 中のアドレスを表すことになる。

4. Label 式の対応 lab。
   関数中の全ての Label 式 (LABEL A s) の $s \in$ String に関数中のロケーション lab$s$ を対応させる。これにより (LABEL A s) は関数中のそのロケーションを表すことになる。

以上のような L メモリと L 環境が具体的に一つあたえられたとすると L 式そして L 関数の意味が確定する。L 式列 $s \in$ Lseq の 1 ステップ実行の意味を $\mathcal{S}[\![s]\!]$ と書くとすると、それは以下のような型をもつ関数である。

$$\mathcal{S}[\![s]\!]: \text{環境} \rightarrow \text{メモリ} \rightarrow \text{メモリ}$$

すなわち $\mathcal{S}[\![s]\!]$ とは、環境を引数とし、直前のメモリから、s の 1 ステップの実行後のメモリを返すステートトランスフォーマを返す関数である。

★他の意味関数についての概説も入れる。

以上の考え方で LIR のセマンティックスを定義していく。意味定義に使われる定数、関数、あるいは型が以下のように定義されている場合。

$$\mathit{foo} \triangleq \text{Unspecified}$$

これは *foo* の具体的な定義を LIR のドキュメントとしては与えない、という意味である。具体的な定義が個々のマシンにより異なるというケースで、一般的には定義することが不可能な場合、定義をこのように Unspecified としておく。ただし、マシンにより異なるとはいっても、意味定義の都合上必要な条件が *foo* に課される場合があるが、それは個別に記述される。このように定義が Unspecified と扱われている定数を以後便宜上 Unspecified 定数と呼ぶ。関数、型についても同様である。これらの具体的な定義はマシン記述の情報から定義される。マシン記述の情報と Unspecified 関数との関係はマシン記述に関するドキュメントで定義される。

5.2 ビット列とバイトの定義

データを表すドメインとして、ビット列を定義する。また、メモリでアドレス指定される最小単位のビット列としてバイトを定義する。現役のハードウエアを考慮し 1 バイトは 8 ビットと定義する。

$$\begin{aligned}
\texttt{Bit} &: \boldsymbol{N} \to \boldsymbol{Type} \\
\texttt{Bit}\,n &\triangleq \{0,1\}^n \quad (= \{b_{n-1}\cdots b_0 \mid b_i \in \{0,1\}\}) \\
\texttt{Bits} &: \boldsymbol{Type} \\
\texttt{Bits} &\triangleq \textstyle\bigcup_{n=0}^{\infty} \texttt{Bit}\,n \\
\# &: \texttt{Bits} \to \boldsymbol{N} \\
\#\,b_{n-1}\cdots b_0 &\triangleq n \\
\# &: \{\texttt{Bit}\,n \mid n \in \boldsymbol{N}\} \to \boldsymbol{N} \\
\#\,\texttt{Bit}\,n &\triangleq n \\
\texttt{Byte} &: \boldsymbol{Type} \\
\texttt{Byte} &\triangleq \texttt{Bit}\,8
\end{aligned}$$

$\texttt{Bit}\,n$ は $\{0,1\}$ の n 直積であり n ビットのデータからなる型を表す。Bits は全ての $n \in \boldsymbol{N}$ についての $\texttt{Bit}\,n$ の和集合であり、意味領域として使用する。$\#b$ で b のビット数を表す。また n ビットのビット列の集合 $\texttt{Bit}\,n$ に対しても $\#\texttt{Bit}\,n$ は n を表すとする。ビット列 $b \in \texttt{Bit}\,n$ 上の操作記述では、その各ビットを並べ $b = b_{n-1}\cdots b_0$ と表記することで各ビットを参照する。$\texttt{Bit}\,0$ は集合論的に 1 点集合 $\{\emptyset\}$ となる。$\emptyset \in \texttt{Bits}$ は実際は何も値を返さないが、形式上 Bits の値を返すとすると意味定義で扱いやすいような関数が返す値として使用される。

以下はビット列を数学的な数と解釈する関数である。$\boldsymbol{N}$ は理想的な符号なし整数、$\boldsymbol{Z}$ は理想的な符号つき整数、$\boldsymbol{R}$ は理想的な浮動小数と考えている。

$$\begin{aligned}
\texttt{nb} &: \texttt{Bits} \to \boldsymbol{N} \\
\texttt{nb}\ b_{n-1}\cdots b_0 &\triangleq \textstyle\sum_{i=0}^{n-1} b_i * 2^i \\
\texttt{zb} &: \texttt{Bits} \to \boldsymbol{Z} \\
\texttt{zb}\ b_{n-1}\cdots b_0 &\triangleq \texttt{nb}\ b_{n-1}\cdots b_0\ - b_{n-1} * 2^n \\
\texttt{rb} &: \texttt{Bits} \to \boldsymbol{R}_{\perp} \\
\texttt{rb}\,b &\triangleq b \text{ の IEEE 浮動小数解釈}
\end{aligned}$$

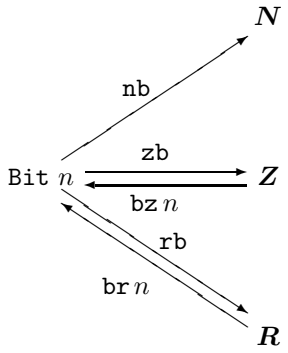
符号付き二進数の表現は過去にいくつか存在したが、現在では 2 の補数表示以外ありえないとの判断で zb は 2 の補数表示を仮定した定義とする。$\mathtt{rb}\,b$ で $b \in \mathtt{Bits}$ が IEEE 解釈で実数とならない場合 $\mathtt{rb}\,b = \bot$ である。以下は数学的な数をビット列に変換する関数である。br も rb 同様部分関数であり、$\mathtt{br}\,n\,x$ で x が IEEE 解釈の n ビット表現を持たない場合は $\mathtt{rb}\,n\,x = \bot$ である。

$$
\begin{aligned}
\mathtt{bz} &: n{:}\boldsymbol{N} \to \boldsymbol{Z} \to \mathtt{Bit}\,n \\
\mathtt{bz}\,n\,x &\triangleq !b \in \mathtt{Bit}\,n \;\; x \equiv \mathtt{nb}\,b \pmod{2^n} \\
\mathtt{br} &: n{:}\boldsymbol{N} \to \boldsymbol{R} \to \mathtt{Bit}\,n_\bot \\
\mathtt{br}\,n\,x &\triangleq x \text{ の } n \text{ ビット IEEE 浮動小数解釈}
\end{aligned}
$$

現在 LIR では整数演算でのオーバーフローは考慮していないが、与えられた整数が n ビットで表現可能かどうかを調べる関数は以下のように定義出来る。将来的にこれらの関数は意味記述で使用されるかもしれない。ここに $\mathtt{isasint}\,n\,x$ は $x \in \boldsymbol{Z}$ が n ビット符号つきで表される数かどうか、また $\mathtt{isauint}\,n\,x$ は $x \in \boldsymbol{N}$ が n ビット符号なしで表される数かどうかを判定する。

$$
\begin{aligned}
\mathtt{isasint} &: \boldsymbol{N} \to \boldsymbol{Z} \to \boldsymbol{Bool} \\
\mathtt{isasint}\,n\,x &\triangleq x = \mathtt{zb}(\mathtt{bz}\,n\,x) \\
\mathtt{isauint} &: \boldsymbol{N} \to \boldsymbol{N} \to \boldsymbol{Bool} \\
\mathtt{isauint}\,n\,x &\triangleq x = \mathtt{nb}(\mathtt{bz}\,n\,x)
\end{aligned}
$$

以上の変換関数を図式としてまとめておく。



以下重要な関数 nb, zb, bz について補足する。 n ビットの 2 の補数表現は数学的にいうと $\boldsymbol{Z}$ の剰余環として得られる単位可換環 $\boldsymbol{Z}/(2^n)$ の代数に対応した、理論的に正しい表現である。ここではビット列 $b \in \mathtt{Bit}\,n$ と $\boldsymbol{Z}/(2^n)$ の剰余類が一対一に対応する。その対応を与えるのが nb, zb, bz であり、b に対して $\mathtt{nb}\,b$ を要素とする $\boldsymbol{Z}/(2^n)$ の剰余類が対応する。言い換えると $\mathtt{nb}\,b$ はその剰余類から代表元 x を $0 \le x < 2^n$ の範囲で選ぶ関数に他ならない。$\mathtt{zb}\,b$ も同様だが代表元 x を $-2^{n-1} \le x < 2^{n-1}$ の範囲で選んでいる点が異なる。bz は整数 x に、それを要素とする剰余類を表すビット列を対応させている。これらの事実を関数の間に成り立つ定理として述べておく。

- 定理 (nb, zb, bz)

1. $\forall n \in \boldsymbol{N}\;\; \forall b \in \mathtt{Bit}\,n \qquad \mathtt{nb}\,b \;\equiv\; \mathtt{zb}\,b \pmod{2^n}$
2. $\forall n \in \boldsymbol{N}\;\; \forall b \in \mathtt{Bit}\,n \qquad \mathtt{bz}\,n\,(\mathtt{zb}\,b) \;=\; b$
3. $\forall n \in \boldsymbol{N}\;\; \forall x \in \boldsymbol{Z} \qquad \mathtt{zb}(\mathtt{bz}\,n\,x) \;\equiv\; x \pmod{2^n}$
4. $\forall n \in \boldsymbol{N}\;\; \forall x, y \in \boldsymbol{Z} \quad \mathtt{bz}\,n\,x = \mathtt{bz}\,n\,y \;\leftrightarrow\; x \equiv y \pmod{2^n}$

3. は 2. と 4. から導けるるが、応用上便利なので上げてある。これらは整数演算式の最適化などで式変形の正当性を証明する際に役に立つと思われる。

★証明の例を入れる。分配律は side effect free の場合のみだが、結合律は OK とかなんとか。

- 議論

Q. LIR では整数演算でオーバーフローを無視しているとのことだが、その理由は。

A. まず第一に COINS で対象とする言語の中で、C と Java はオーバーフローを無視している (直接書かれた定数についてはチェックしているが) という理由が上げられる。FORTRAN については良く知らないが多分無視ではないか。
第二にオーバーフローを無視する LIR の整数演算は副作用を無視すれば単位可換環をなすことが証明できる。(オーバーフローを無視しないと、そういう数学的に奇麗な性質は成り立たなくなる) すなわち環の演算のみからなる式は数学的に許される自由な変形をおこなっても意味は変わらない。したがってもし「コンパイラは入力がオーバーフローや未定義値を発生させないと仮定し、その範囲で最適化変換を行なって良い」という前提が許されるならば、効率的な最適化を安心して行なうことが可能となる。この前提は少なくとも COINS が対象とする言語では事実上認められていると思われる。
まとめると、まずオーバーフローを無視しない演算のみに統一することには賛成出来ない。もしそれが必要なら今のオーバーフローを無視する演算に加える形にすべきである。オーバーフローを無視しない整数演算が必要となった時には容易に仕様追加することは出来るが、今は入れていない。もし追加するとしたら、オーバーフローをチェックする、というモディファイアを必要な整数演算に追加することになるだろう。

5.3 L タイプのセマンティックス ($\mathcal{T}$)

$\texttt{Ltype} = \texttt{Itype} + \texttt{Ftype} + \texttt{Fixnum}$ である。型 t が I または F で始まる十進数のとき、その値を t のビット幅とよぶ。Fixnum の場合はそれがビット幅である。制約 validltype によって、それは 8 で割り切れる正の整数である。

$\mathcal{T}$ は L タイプの意味関数であり、与えられた L タイプの意味としての型を返す。直観的には $\mathcal{T}[\![\texttt{I}n]\!]$ は n ビットの整数型、$\mathcal{T}[\![\texttt{F}n]\!]$ は n ビットの浮動小数型を表すと考えたいかも知れない。しかし、我々の定義では以下に示すように $\texttt{I}n$ と $\texttt{F}n$ に同一のセマンティックスを与えている。

$$
\begin{aligned}
\mathcal{T} &: \texttt{Ltype} \to \boldsymbol{Type} \\
\mathcal{T}[\![\texttt{I}n]\!] &\triangleq \texttt{Bit}\,n \\
\mathcal{T}[\![\texttt{F}n]\!] &\triangleq \texttt{Bit}\,n \\
\mathcal{T}[\![n]\!] &\triangleq \texttt{Bit}\,n \\
\texttt{Bitw} &: \boldsymbol{Type} \\
\texttt{Bitw} &\triangleq \{n \in \boldsymbol{N} \mid \exists t \in \texttt{Ltype}\ \mathcal{T}[\![t]\!] = \texttt{Bit}\,n\}
\end{aligned}
$$

符号つきであるか符号なしであるか、あるいは浮動小数であるか、という区別はタグつきアーキテクチャのようなものを除き、COINS で対象としているハードではデータそのものの型ではなく本来は演算で区別されるものである。セマンティックスの定義でもそれを反映し、単にそのビット幅 n を持つ $\texttt{Bit}\,n$ 型と扱う。Bitw は意味記述においてビット幅の型であることを明示するために使われる。

5.4 L メモリの定義

L メモリは関数カウンタ、レジスタメモリ、データメモリからなる。バイト単位でアドレスが付いているものとする。すなわちメモリ m とは $\boldsymbol{N}$ から Byte への部分関数 $m\colon \boldsymbol{N} \to \texttt{Byte}_{\perp}$ と考える。よって $\texttt{dom}\,m$ が実際のアドレス空間を表す。このアドレス空間は後に説明するようにフレームのアロケーションにより拡張される。

以下はロケーション、プログラムメモリのアドレス、レジスタメモリのアドレス、そしてデータメモリのアドレスの型定義である。実際には全て $\boldsymbol{N}$ であり、この定義は単にドキュメント上の便宜である。

$$
\begin{aligned}
\mathtt{Location}, \mathtt{PMemAddr}, \mathtt{RMemAddr}, \mathtt{DMemAddr} \ &: \ \boldsymbol{Type} \\
\mathtt{Location} &\triangleq \boldsymbol{N} \\
\mathtt{PMemAddr} &\triangleq \boldsymbol{N} \\
\mathtt{RMemAddr} &\triangleq \boldsymbol{N} \\
\mathtt{DMemAddr} &\triangleq \boldsymbol{N}
\end{aligned}
$$

以下はプログラムメモリの型 PMem、レジスタメモリの型 RMem、及びデータメモリの型 DMem の定義である。

$$
\begin{aligned}
\mathtt{PMem}, \mathtt{RMem}, \mathtt{DMem} \ &: \ \boldsymbol{Type} \\
\mathtt{PMem} &\triangleq \mathtt{PMemAddr} \to \mathtt{Lfunc}_\perp \\
\mathtt{RMem} &\triangleq \mathtt{RMemAddr} \to \mathtt{Byte}_\perp \\
\mathtt{DMem} &\triangleq \mathtt{DMemAddr} \to \mathtt{Byte}_\perp
\end{aligned}
$$

意味記述におけるメモリの扱いで、やや技巧的なものがあるので、ここで説明しておく。意味記述ではメモリが部分関数として表現されていることを利用して L 関数の実行中にのみ存在するフレームを表現する。たとえば m:DMem というメモリに L 関数の実行により生まれたフレームメモリ m':DMem (ここに $\mathtt{dom}\, m' \cap \mathtt{dom}\, m = \emptyset$) を追加したものは $m'' \triangleq m :\!= m'$ と表される。L 関数の実行においてはフレームもそれ以外の部分も修正されるが、L 関数終了時にはフレームが解放され、もとの m のうちフレーム以外の部分の修正のみが L 関数の副作用として記録されなければならない。それはすなわち m'' のドメインを元の $\mathtt{dom}\, m$ に制限したものに他ならない。つまり L 関数の副作用は $m''|_{\mathtt{dom}\, m}$ と表現出来る。同様のテクニックはレジスタメモリにも適用されている。詳細は L 関数のセマンティックスを参照のこと。

メモリへの参照により IO を行なうようなマシンでは関数の意味に特定のメモリへの書き込み、読み込みを時系列として記録したリストを含める必要がある。このような外部とのインタラクションを表現するために、アクションの時系列情報を記録するトレースというリストが L メモリには存在する。

$$
\begin{aligned}
\mathtt{Action}, \mathtt{Trace} \ &: \ \boldsymbol{Type} \\
\mathtt{Action} &\triangleq \mathtt{Symbol} \times \mathtt{Ltype} \times [\mathtt{Bits}] \\
\mathtt{Trace} &\triangleq [\mathtt{Action}]
\end{aligned}
$$

アクションのリスト型が Trace である。以下は現在定義されているアクション、そのアクションを直接記録する (可能性のある) L 式のコード、そのアクションの表現である。

アクション	Action
volatile メモリ読み込み	(READ, $ltype$, $[address]$)
volatile メモリ書き込み	(WRITE, $ltype$, $[address, value]$)

以上をまとめたものが L メモリとなる。以下の Mem が L メモリの型である。最後の rs はランダムステートである。

$$
\begin{aligned}
\mathtt{Mem} \ &: \ \boldsymbol{Type} \\
\mathtt{Mem} &\triangleq \{\mathtt{pc}\colon \mathtt{Location},\ \mathtt{pm}\colon \mathtt{PMem},\ \mathtt{rm}\colon \mathtt{RMem},\ \mathtt{dm}\colon \mathtt{DMem},\ \mathtt{tr}\colon \mathtt{Trace},\ \mathtt{rs}\colon \boldsymbol{R}\}
\end{aligned}
$$

意味記述の簡潔さのために、アクションをトレースに追加する関数を定義する。

$$\begin{aligned} \texttt{addtotr} &: \texttt{Mem} \to \texttt{Symbol} \to \texttt{Ltype} \to [\texttt{Bits}] \to \texttt{Mem} \\ \texttt{addtotr}\,\sigma\,s\,[\![t]\!]\,b &\triangleq \sigma := \{.\texttt{tr} = \sigma.\texttt{tr} +\!\!+ [(s, [\![t]\!], b)]\} \end{aligned}$$

メモリの Endianness や境界条件はマシンに依存するので、メモリへの読み書き関数を Unspecified な部分関数として定義する。

$$\begin{aligned} \texttt{bits2bytes} &: \texttt{Bit}(n * \#\,\texttt{Byte}) \to [\texttt{Byte}] \\ \texttt{bits2bytes}\,w_{n-1}\cdots w_1 w_0 &\triangleq [w_0, w_1, \ldots, w_{n-1}] \\ \texttt{bytes2bits} &: l{:}\,[\texttt{Byte}] \to \texttt{Bit}(\#\,l * \#\,\texttt{Byte}) \\ \texttt{bytes2bits}\,[w_0, w_1, \ldots, w_{n-1}] &\triangleq w_{n-1}\cdots w_1 w_0 \\ \texttt{readbytes} &: (\boldsymbol{N} \to \texttt{Byte}) \to \boldsymbol{N} \to \boldsymbol{N} \to [\texttt{Byte}] \\ \texttt{readbytes}\,m\,l\,n &\triangleq [ml, m(l+1), \ldots, m(l+n-1)] \\ \texttt{writebytes} &: (\boldsymbol{N} \to \texttt{Byte}) \to \boldsymbol{N} \to \boldsymbol{N} \to [\texttt{Byte}] \to (\boldsymbol{N} \to \texttt{Byte}) \\ \texttt{writebytes}\,m\,l\,n\,[b_0, b_1, \ldots, b_{n-1}] &\triangleq m := [l \mapsto b_0, l+1 \mapsto b_1, \ldots, l+n-1 \mapsto b_{n-1}] \end{aligned}$$

bits2bytes と bytes2bits はワードと Byte のリストの相互変換である。readbytes $m\,l\,n$ はメモリ m のアドレス l から n バイトをリストの形で読む。writebytes $m\,l\,b$ はメモリ m のアドレス l からバイトのリスト b で与えられたバイトを順時書き込む。(ただし数学的な関数として定義するために、破壊的に書き込むのではなく、対応を修正したメモリを返す、という形で定義している。)

以下はメモリやレジスタアクセスに関する L 式の意味定義で使われる関数である。これらはトレースにはアクセスしない。

$$\begin{aligned} \texttt{rmread} &: \texttt{Mem} \to \texttt{RMemAddr} \to t{:}\,\texttt{Ltype} \to \mathcal{T}[\![t]\!]_{\perp} \\ \texttt{rmread} &\triangleq \text{Unspecified} \\ \texttt{dmread} &: \texttt{Mem} \to \texttt{DMemAddr} \to t{:}\,\texttt{Ltype} \to \mathcal{T}[\![t]\!]_{\perp} \\ \texttt{dmread} &\triangleq \text{Unspecified} \\ \texttt{rmwrite} &: \texttt{Mem} \to \texttt{RMemAddr} \to t{:}\,\texttt{Ltype} \to \mathcal{T}[\![t]\!] \to \texttt{Mem}_{\perp} \\ \texttt{rmwrite} &\triangleq \text{Unspecified} \\ \texttt{dmwrite} &: \texttt{Mem} \to \texttt{DMemAddr} \to t{:}\,\texttt{Ltype} \to \mathcal{T}[\![t]\!] \to \texttt{Mem}_{\perp} \\ \texttt{dmwrite} &\triangleq \text{Unspecified} \end{aligned}$$

rmread $\sigma\,a\,[\![t]\!]$ はレジスタメモリ σ.rm のアドレス a にある型 t のオブジェクトの値を型 $\mathcal{T}[\![t]\!]$ の値として返す。dmread もデータメモリに対する同様の関数である。rmwrite $\sigma\,a\,[\![t]\!]\,w$ はレジスタメモリ σ.rm のアドレス a に、型 t の値 w を書き込む (もちろん正確に言えば σ は破壊しないで、修正された新たなメモリを返す)。dmwrite も同様である。

dmread と dmwrite は部分関数である。これにより有限なメモリ、バウンダリ条件を表現する。

具体的なマシンに対する記述例として VAX に代表される Little Endian マシンの場合の dmread, dmwrite の定義を上げる。混乱をさけるために、名前の後ろに VAX を付ける。この定義ではメモリは無限で、境界条件もないとしている。

$$\begin{aligned} \texttt{dmreadVAX}\,\sigma\,a\,[\![t]\!] &\triangleq \texttt{bytes2bits}(\texttt{readbytes}\,\sigma.\texttt{dm}\;a\,(\#\,\mathcal{T}[\![t]\!]/8)) \\ \texttt{dmwriteVAX}\,\sigma\,a\,[\![t]\!]\,w &\triangleq \sigma := \{.\texttt{dm} = \texttt{writebytes}\,\sigma.\texttt{dm}\;a\,(\#\,\mathcal{T}[\![t]\!]/8)\,(\texttt{bits2bytes}\,w)\} \end{aligned}$$

IBM に代表される Big Endian マシンでは次のように reverse を入れなければならない。

$$\begin{aligned}
\texttt{dmreadIBM}\,\sigma\,a\,[\![t]\!] &\triangleq \texttt{bytes2bits}(\texttt{reverse}\,(\texttt{readbytes}\,\sigma.\texttt{dm}\,a\,(\#\mathcal{T}[\![t]\!]/8))\\
\texttt{dmwriteIBM}\,\sigma\,a\,[\![t]\!]\,w &\triangleq \sigma := \{.\texttt{dm} = \texttt{writebytes}\,\sigma.\texttt{dm}\,a\,(\#\mathcal{T}[\![t]\!]/8)\,(\texttt{reverse}\,(\texttt{bits2bytes}\,w))\}
\end{aligned}$$

Cross Endian マシン、あるいはレジスタメモリとデータメモリで Endianness が異なるというようなマシンがあるかも知れないので、読み書き関数はレジスタメモリとデータメモリを分けている。整数と浮動小数で Endianness が異なるというようなマシンでも、引数に L タイプを取っているので表現は可能である。

メモリ関連の定義の最後に「乱数発生関数」を定義する。乱数は意味記述において、 なにか不定な値 (⊥ ではない) が代入されるということを表すため、及びある特定の値を予測することで可能となるような (あやまった) 最適化を禁止するために使われる。

乱数発生関数 random を数学的な関数として定義するために、L メモリのランダムステートの値から何らかの方法で「乱数」と「ランダムステートの値を次の状態に変更した L メモリ」を返すものとする。この関数で作られた乱数を**予測不能値**という。意味記述の便宜のために random 自体は以下のように引数として L メモリを取るとし、本質的な乱数関数は rand とする。そして rand は Unspecified 関数とする。その意図はそのインプリメントを最適化ルーチンに予測出来ないようにすることである。

$$\begin{aligned}
\texttt{random} &: t{:}\,\texttt{Ltype} \to \texttt{Mem} \to \texttt{Mem} \times \mathcal{T}[\![t]\!]\\
\texttt{random}[\![t]\!]\,\sigma &\triangleq (\sigma := \{.\texttt{rs} = r\},\, v)\\
&\quad\ \textbf{where}\ (r, v) \triangleq \texttt{rand}[\![t]\!]\,\sigma.\texttt{rs}\\
\texttt{rand} &: t{:}\,\texttt{Ltype} \to \boldsymbol{R} \to \boldsymbol{R} \times \mathcal{T}[\![t]\!]\\
\texttt{rand} &\triangleq \text{Unspecified}
\end{aligned}$$

5.5 L 環境の定義

L 環境は L 式と Mem の対応である。Frame 式の引数文字列が対応するのはオフセット値であり、その値とフレームポインタの値の和が Frame 式の表すものであった。フレームの伸びる向きはインプリメント依存なので、オフセットの型は $\boldsymbol{Z}$ とする。

$$\begin{aligned}
\texttt{FrameOffset} &: \boldsymbol{Type}\\
\texttt{FrameOffset} &\triangleq \boldsymbol{Z}
\end{aligned}$$

以下は L 環境を構成するテーブルの型の定義である。

$$\begin{aligned}
\texttt{RegEnv}, \texttt{StaticEnv}, \texttt{FrameEnv}, \texttt{LabelEnv} &: \boldsymbol{Type}\\
\texttt{RegEnv} &\triangleq \texttt{String} \to \texttt{RMemAddr}_\perp\\
\texttt{StaticEnv} &\triangleq \texttt{String} \to \texttt{DMemAddr}_\perp\\
\texttt{FrameEnv} &\triangleq \texttt{String} \to \texttt{FrameOffset}_\perp\\
\texttt{LabelEnv} &\triangleq \texttt{String} \to \texttt{Location}_\perp
\end{aligned}$$

このように定義上は全てのテーブルを全関数として定義する。すなわち L 関数に現れてもいない全てのシンボルや文字列について何らかの割り当てを行なっているものをテーブルとする。これらテーブルを集めたものが、L 環境となる。以下の Env が L 環境の型である。

$$\begin{aligned}
&\texttt{Env} \;:\; \boldsymbol{Type}\\
&\texttt{Env} \triangleq \{\texttt{reg}\colon \texttt{RegEnv},\ \texttt{sta}\colon \texttt{StaticEnv},\ \texttt{fra}\colon \texttt{FrameEnv},\ \texttt{lab}\colon \texttt{LabelEnv}\}
\end{aligned}$$

以下は L 式列からラベル環境を作る関数 mklabelenv の「つまらない定義」である。L 式列中の DefLabel 式の文字列にそのロケーションを対応させる部分関数を返す。ただし二重定義が存在したらシンボル DUP を返す。正しい L 式列ではこのようなことは起きない (実は validseq の定義にこの関数を使っている)。

ラベル環境については L 式列から直ちに定まる。にもかかわらず LIR の表現ではラベル文字列を介した表現にした理由は人間にとっての見やすさを考慮してのことであり、インプリメントではラベルの参照を直接データ構造のリンクで表現することも考えられる。

$$\begin{aligned}
&\texttt{mklabelenv} \;:\; \texttt{Lseq} \to \texttt{LabelEnv} + \{\texttt{DUP}\}\\
&\texttt{mklabelenv}\, s \triangleq \texttt{m}\, s\, 0\\
&\qquad \textbf{where}\\
&\qquad\qquad \texttt{m} \;:\; \texttt{Lseq} \to \boldsymbol{N} \to \texttt{LabelEnv} + \{\texttt{DUP}\}\\
&\qquad\qquad \texttt{m}\,[\![()]\!]\, i \triangleq \bot_{(\texttt{String}\to N_\bot)} \qquad (=\ \lambda s.\bot_N)\\
&\qquad\qquad \texttt{m}\,[\![((\texttt{DEFLABEL}\, s)\ e_1\ \cdots\ e_n)]\!]\, i \triangleq \textbf{if } l = \texttt{DUP} \vee l\, s \neq \bot \textbf{ then DUP else } l := s \mapsto i\\
&\qquad\qquad\qquad\qquad \textbf{where } l \triangleq \texttt{m}\,[\![(e_1\ \cdots\ e_n)]\!]\,(i+1)\\
&\qquad\qquad\qquad \texttt{m}\,[\![(e_0\ e_1\ \cdots\ e_n)]\!]\, i \triangleq \texttt{m}\,[\![(e_1\ \cdots\ e_n)]\!]\,(i+1) \quad \textbf{if } e_0 \notin \texttt{DefLabelExp}
\end{aligned}$$

5.6 L 式のセマンティックス $(\mathcal{S}, \mathcal{E})$

$\mathcal{S}$ が L 式列 s の 1 ステップ実行の意味関数である。$\mathcal{S}[\![s]\!]$ へ具体的な環境 ρ: Env を引数として与えると 1 ステップ実行を行なう Mem 間のステートトランスフォーマを返す。

$$\begin{aligned} \mathcal{S} &: \mathtt{Lseq} \to \mathtt{Env} \to \mathtt{Mem} \to \mathtt{Mem}_\perp \\ \mathcal{S}[\![s]\!]\rho\,\sigma &\triangleq \sigma_2 \quad \textbf{where} \\ \sigma_1 &\triangleq \sigma := \{.\mathtt{pc} = \sigma.\mathtt{pc} + 1\} \\ (\sigma_2, v) &\triangleq \mathcal{E}[\![s\,!\,\sigma.\mathtt{pc}]\!]\rho\,\sigma_1 \end{aligned}$$

ここで $\mathcal{E}$ はひとつの L 式の意味関数であり、変更された Mem と式の値の組を返す。

$$\mathcal{E} : \mathtt{Lexp} \to \mathtt{Env} \to \mathtt{Mem} \to (\mathtt{Mem} \times \mathtt{Bits})_\perp$$

$\mathcal{E}$ は UnTypedExp にも適用される。その場合、値は $\emptyset$ となる。以下、各 L 式に対してその意味記述を行なう。また同時に validexp_sem も定義される。以下は意味記述の例である:

$(\mathtt{CONVSX}\ t\ x_1) \qquad t, t_1 \in \mathtt{Itype} \ \wedge\ w > w_1$
符号拡張。

$$\mathcal{E}[\![(\mathtt{CONVSX}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \mathtt{bz}\,w\,(\mathtt{zb}\,v_1))$$

各意味記述において、簡単のために次の規約を置く。記述に現れる x_i が $x_i \in$ TypedExp で、記号 $t_i, w, w_i, \sigma_i, v_i$ がその記述では定義されていない場合、以下の定義がその記述内で使用出来るとする。尚、この定義は副作用のある引数が通常は出現順に左から右へと評価されることを意味している。

$$\begin{aligned} t_i &\triangleq x_i.\mathtt{type} \\ w &\triangleq \#\mathcal{T}[\![t]\!] \\ w_i &\triangleq \#\mathcal{T}[\![t_i]\!] \\ (\sigma_1, v_1) &\triangleq \mathcal{E}[\![x_1]\!]\rho\sigma \\ (\sigma_2, v_2) &\triangleq \mathcal{E}[\![x_2]\!]\rho\sigma_1 \\ &\vdots \\ (\sigma_i, v_i) &\triangleq \mathcal{E}[\![x_i]\!]\rho\sigma_{i-1} \end{aligned}$$

上記意味記述の左上の L 式を e とする。実際は e のさらに左端に参照番号が現れるが、それは意味記述の部分ではない。まず記述内に定義の無い $t_1, w, w_1, \sigma_1, v_1$ が先の規約に従い定義される。e に続く制約条件が validexp_sem$e$ となる。この場合は以下の定義がこの意味記述に付随しているとする。

$$\mathtt{validexp\_sem}\,[\![(\mathtt{CONVSX}\ t\ x_1)]\!] \triangleq t, t_1 \in \mathtt{Itype} \ \wedge\ w > w_1$$

制約条件がない場合は validexp_sem$e \triangleq$ **true** である。その次に言葉による説明と、$\mathcal{E}[\![e]\!]$ のフォーマルな定義が与えられる。ある意味記述内で、その意味定義に必要な Unspecified 関数を定義することがある。また複数の意味記述で共有できるような補助関数を定義することもある。これらの関数のスコープは本ドキュメント全体である。意味関数は L 式のパターンごとに分けて定義されているが、どのパターンにもマッチしない L 式 e については validexp_sem$e \triangleq$ **false** とする。

5.6.1 Const 式

ここでは Const 式に対する $\mathcal{E}$ を定義する。ASMCONST 式は Pure 式に分類されている。

1. $(\texttt{INTCONST}\ t\ z)$ $\qquad t \in \texttt{Itype}$

   $z \in \boldsymbol{Z}$ を型 t で表した整数値を表す。

$$\mathcal{E}[\![(\texttt{INTCONST}\ t\ z)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{bz}\,w\,z)$$

2. $(\texttt{FLOATCONST}\ t\ r)$ $\qquad t \in \texttt{Ftype} \wedge \texttt{br}\,w\,r \neq \bot$

   $r \in \boldsymbol{R}$ を型 t で表した浮動小数値を表す。

$$\mathcal{E}[\![(\texttt{FLOATCONST}\ t\ r)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{br}\,w\,r)$$

いくつかの意味記述では記述内で Const 式を作り、記述に使用しているので、Const 式を作る関数をここで定義しておく。

$$\begin{aligned}
\texttt{mkconst} \ &: \ t{:}\texttt{Ltype} \to \mathcal{T}[\![t]\!] \to \texttt{Lexp} \\
\texttt{mkconst}[\![t]\!]\,b \ &\triangleq \ \textbf{if}\ t \in \texttt{Itype}\ \textbf{then} \\
&\qquad [\![(\texttt{INTCONST}\ t\ z)]\!] \quad \textbf{where}\ z \triangleq \texttt{zb}\,b \\
&\quad \textbf{else} \\
&\qquad [\![(\texttt{FLOATCONST}\ t\ r)]\!] \quad \textbf{where}\ r \triangleq \texttt{rb}\,b
\end{aligned}$$

5.6.2 Addr 式

ここでは Addr 式に対する $\mathcal{E}$ を定義する。Static 式はグローバル変数のアドレス、Frame 式はローカル変数のアドレスを表すためのものである。Label 式は L 関数のロケーションを表す。

1. $(\texttt{STATIC}\ t\ s)$ $\qquad t \in \texttt{Itype}$

   $s \in \texttt{String}$ で示されるプログラムメモリまたはデータメモリのアドレス。

$$\mathcal{E}[\![(\texttt{STATIC}\ t\ s)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{bz}\,w\,(\rho.\texttt{sta}[\![s]\!]))$$

2. $(\texttt{FRAME}\ t\ s)$ $\qquad t \in \texttt{Itype}$

   $s \in \texttt{String}$ で示されるオフセットとフレームポインタの和で生成されるデータメモリのアドレス。

$$\mathcal{E}[\![(\texttt{FRAME}\ t\ s)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{bz}\,w\,(\texttt{nb}(\texttt{rmread}\,\sigma\,(\rho.\texttt{reg}[\![\texttt{"@FP"}]\!])\,[\![t]\!]) + \rho.\texttt{fra}[\![s]\!]))$$

3. $(\texttt{LABEL}\ t\ s)$ $\qquad t \in \texttt{Itype}$

   $s \in \texttt{String}$ で示されるロケーション。

$$\mathcal{E}[\![(\texttt{LABEL}\ t\ s)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{bz}\,w\,(\rho.\texttt{lab}[\![s]\!]))$$

5.6.3 Reg 式

ここでは Reg 式に対する $\mathcal{E}$ を定義する。マシンの自然なワード長より小さな単位での演算等で、オペランドに指定されたレジスタの一部にアクセスする場合がある。そのような部分的なレジスタを、元のレジスタの**部分レジスタ**という。

以下の定義が使われるのは Reg 式が Set 式の第一引数以外で、そのレジスタが格納している値を取り出すという場合にかぎられるように Set 式が定義されている。

1. $(\texttt{REG}\ t\ s)$

   $s \in \texttt{String}$ で示されるレジスタの値。

$$\mathcal{E}[\![(\texttt{REG}\ t\ s)]\!]\rho\,\sigma \triangleq (\sigma, \texttt{rmread}\,\sigma\,(\rho.\texttt{reg}[\![s]\!])\,[\![t]\!])$$

2. $(\texttt{SUBREG}\ t\ (\texttt{REG}\ t_1\ s)\ n\ [\&\ m])\quad w < w_1\ \wedge\ 0 \le n < w_1/w \in \boldsymbol{N}$

   第一引数で指定されるレジスタの n 番目の部分レジスタの値。$m \in \texttt{SubregModifier}$ はここでは参照されない。このモディファイアの意味については Set 式の意味記述を参照のこと。

$$\mathcal{E}[\![(\texttt{SUBREG}\ t\ (\texttt{REG}\ t_1\ s)\ n\ [\&\ m])]\!]\rho\,\sigma \triangleq (\sigma, \texttt{rmread}\,\sigma(\rho.\texttt{reg}[\![s]\!] + \texttt{subregoffset}[\![t]\!]\,[\![t_1]\!]\,n)\,[\![t]\!])$$

   ここに subregoffset は型 t_1 のレジスタ中で n 番目の型 t の部分レジスタのオフセットを与える Unspecified 関数である。

$$\begin{aligned}&\texttt{subregoffset}\ :\ \texttt{Ltype} \to \texttt{Ltype} \to \boldsymbol{N} \to \boldsymbol{N}\\ &\texttt{subregoffset} \triangleq \text{Unspecified}\end{aligned}$$

   ★これでは一般的すぎる。

5.6.4 Pure 式

ここでは Pure 式に対する $\mathcal{E}$ を定義する。

1. $(\texttt{NEG}\ t\ x_1)\qquad t = t_1$

   符号反転。

$$\begin{aligned}\mathcal{E}[\![(\texttt{NEG}\ t\ x_1)]\!]\rho\,\sigma &\triangleq (\sigma_1, \texttt{bz}\,w\,(-\texttt{zb}\,v_1)) \quad \textbf{if}\ t \in \texttt{Itype}\\ \mathcal{E}[\![(\texttt{NEG}\ t\ x_1)]\!]\rho\,\sigma &\triangleq (\sigma_1, \texttt{br}\,w\,(-\texttt{rb}\,v_1)) \quad \textbf{if}\ t \in \texttt{Ftype}\end{aligned}$$

2. $(\texttt{ADD}\ t\ x_1\ x_2)\qquad t = t_1 = t_2$

   加算。

$$\begin{aligned}\mathcal{E}[\![(\texttt{ADD}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{bz}\,w\,(\texttt{zb}\,v_1 + \texttt{zb}\,v_2)) \quad \textbf{if}\ t \in \texttt{Itype}\\ \mathcal{E}[\![(\texttt{ADD}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{br}\,w\,(\texttt{rb}\,v_1 + \texttt{rb}\,v_2)) \quad \textbf{if}\ t \in \texttt{Ftype}\end{aligned}$$

3. $(\texttt{SUB}\ t\ x_1\ x_2)\qquad t = t_1 = t_2$

   減算。

$$\begin{aligned}\mathcal{E}[\![(\texttt{SUB}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{bz}\,w\,(\texttt{zb}\,v_1 - \texttt{zb}\,v_2)) \quad \textbf{if}\ t \in \texttt{Itype}\\ \mathcal{E}[\![(\texttt{SUB}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{br}\,w\,(\texttt{rb}\,v_1 - \texttt{rb}\,v_2)) \quad \textbf{if}\ t \in \texttt{Ftype}\end{aligned}$$

4. $(\texttt{MUL}\ t\ x_1\ x_2)$ $\quad t = t_1 = t_2$

乗算。

$$\mathcal{E}[\![(\texttt{MUL}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{bz}\,w\,(\texttt{zb}\,v_1 * \texttt{zb}\,v_2)) \quad \textbf{if } t \in \texttt{Itype}$$
$$\mathcal{E}[\![(\texttt{MUL}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{br}\,w\,(\texttt{rb}\,v_1 * \texttt{rb}\,v_2)) \quad \textbf{if } t \in \texttt{Ftype}$$

5. $(\texttt{DIVS}\ t\ x_1\ x_2)$ $\quad t = t_1 = t_2$

符号つき除算。

$$\mathcal{E}[\![(\texttt{DIVS}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{divz}\,w\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2)) \quad \textbf{if } t \in \texttt{Itype}$$
$$\mathcal{E}[\![(\texttt{DIVS}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{divr}\,w\,(\texttt{rb}\,v_1)\,(\texttt{rb}\,v_2)) \quad \textbf{if } t \in \texttt{Ftype}$$
$$\texttt{divz} \ : \ w{:}\texttt{Bitw} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\,w_\perp$$
$$\texttt{divz}\,w\,n\,d \triangleq \textbf{if } d = 0 \textbf{ then } \perp \textbf{ else } \texttt{bz}\,w\,(\texttt{truncate}(n/d))$$
$$\texttt{divr} \ : \ w{:}\texttt{Bitw} \to \boldsymbol{R} \to \boldsymbol{R} \to \texttt{Bit}\,w_\perp$$
$$\texttt{divr}\,w\,n\,d \triangleq \textbf{if } d = 0 \textbf{ then } \perp \textbf{ else } \texttt{br}\,w\,(n/d)$$

6. $(\texttt{DIVU}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1 = t_2$

符号なし除算。

$$\mathcal{E}[\![(\texttt{DIVU}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{divz}\,w\,(\texttt{nb}\,v_1)\,(\texttt{nb}\,v_2))$$

7. $(\texttt{MODS}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1 = t_2$

符号つき剰余。

$$\mathcal{E}[\![(\texttt{MODS}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{modz}\,w\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2))$$
$$\texttt{modz} \ : \ w{:}\texttt{Bitw} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\,w_\perp$$
$$\texttt{modz}\,w\,n\,d \triangleq \textbf{if } d = 0 \textbf{ then } \perp \textbf{ else } \texttt{bz}\,w\,(n - d * \texttt{truncate}(n/d))$$

8. $(\texttt{MODU}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1 = t_2$

符号なし剰余。

$$\mathcal{E}[\![(\texttt{MODU}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{modz}\,w\,\sigma_2\,(\texttt{nb}\,v_1)\,(\texttt{nb}\,v_2))$$

9. $(\texttt{CONVSX}\ t\ x_1)$ $\quad t, t_1 \in \texttt{Itype} \ \wedge\ w > w_1$

符号拡張。

$$\mathcal{E}[\![(\texttt{CONVSX}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{zb}\,v_1))$$

10. $(\texttt{CONVZX}\ t\ x_1)$ $\quad t, t_1 \in \texttt{Itype} \ \wedge\ w > w_1$

ゼロ拡張。

$$\mathcal{E}[\![(\texttt{CONVZX}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{nb}\,v_1))$$

11. $(\texttt{CONVIT}\ t\ x_1)$ $\quad t, t_1 \in \texttt{Itype} \ \wedge\ w < w_1$

精度の低い整数へ。

$$\mathcal{E}[\![(\texttt{CONVIT}\ t\ x_1)]\!]\rho\,\sigma \triangleq (\sigma_1, \texttt{bz}\,w\,(\texttt{zb}\,v_1))$$

12. $(\texttt{CONVFX}\ t\ x_1)$ $\quad t, t_1 \in \texttt{Ftype} \ \wedge\ w > w_1$

精度の高い浮動小数へ。

$$\mathcal{E}[\![(\texttt{CONVFX}\ t\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_1, \texttt{br}\,w\,(\texttt{rb}\,v_1))$$

13. $(\texttt{CONVFT}\ t\ x_1)$ $\quad t, t_1 \in \texttt{Ftype} \ \wedge\ w < w_1$

精度の低い浮動小数へ。

$$\mathcal{E}[\![(\texttt{CONVFT}\ t\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_1, \texttt{br}\,w\,(\texttt{rb}\,v_1))$$

14. $(\texttt{CONVFI}\ t\ x_1)$ $\quad t \in \texttt{Itype} \ \wedge\ t_1 \in \texttt{Ftype}$

浮動小数から整数へ。

$$\mathcal{E}[\![(\texttt{CONVFI}\ t\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_1, \texttt{bz}\,w\,(\texttt{truncate}(\texttt{rb}\,v_1)))$$

15. $(\texttt{CONVSF}\ t\ x_1)$ $\quad t \in \texttt{Ftype} \ \wedge\ t_1 \in \texttt{Itype}$

符号つき整数から浮動小数へ。

$$\mathcal{E}[\![(\texttt{CONVSF}\ t\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_1, \texttt{br}\,w\,(\texttt{zb}\,v_1))$$

16. $(\texttt{CONVUF}\ t\ x_1)$ $\quad t \in \texttt{Ftype} \ \wedge\ t_1 \in \texttt{Itype}$

符号なし整数から浮動小数へ。

$$\mathcal{E}[\![(\texttt{CONVUF}\ t\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_1, \texttt{br}\,w\,(\texttt{nb}\,v_1))$$

17. $(\texttt{BAND}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1 = t_2$

論理積。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{BAND}\ t\ x_1\ x_2)]\!]\rho\,\sigma \ &\stackrel{\triangle}{=}\ (\sigma_2, \texttt{bitop}\,(\lambda xy.\textbf{if}\ x = 1 \wedge y = 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\,v_1\,v_2)\\
\texttt{bitop}\ \ &:\ \ (\texttt{Bit}\,1 \to \texttt{Bit}\,1 \to \texttt{Bit}\,1) \to \texttt{Bit}\,n \to \texttt{Bit}\,n \to \texttt{Bit}\,n\\
\texttt{bitop}\,f\ x_{n-1}\cdots x_0\ y_{n-1}\cdots y_0 \ &\stackrel{\triangle}{=}\ z_{n-1}\cdots z_0 \quad \textbf{where}\ \ z_i \stackrel{\triangle}{=} f\,x_i\,y_i
\end{aligned}$$

18. $(\texttt{BOR}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1 = t_2$

論理和。

$$\mathcal{E}[\![(\texttt{BOR}\ t\ x_1\ x_2)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_2, \texttt{bitop}\,(\lambda xy.\textbf{if}\ x = 1 \vee y = 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\,v_1\,v_2)$$

19. $(\texttt{BXOR}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1 = t_2$

排他的論理和。

$$\mathcal{E}[\![(\texttt{BXOR}\ t\ x_1\ x_2)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_2, \texttt{bitop}\,(\lambda xy.\textbf{if}\ x = y\ \textbf{then}\ 0\ \textbf{else}\ 1)\,v_1\,v_2)$$

20. $(\texttt{BNOT}\ t\ x_1)$ $\quad t \in \texttt{Itype} \ \wedge\ t = t_1$

論理否定。

$$\mathcal{E}[\![(\texttt{BNOT}\ t\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ (\sigma_1, \texttt{bitop}\,(\lambda xy.\textbf{if}\ x = 1\ \textbf{then}\ 0\ \textbf{else}\ 1)\,v_1\,(\texttt{bz}\,w\,0))$$

21. $(\texttt{LSHS}\ t\ x_1\ x_2\ [\&\ m])$ $\quad t, t_2 \in \texttt{Itype}\ \wedge\ t = t_1$

符号つき左シフト。x_2 の値とモディファイア $m = (s\ n\ d)$ に従い、以下のようにシフトカウントが決まる。まず x_2 の値を n ビットの符号つき ($s = \texttt{S}$)、あるいは符号なし ($s = \texttt{U}$) 整数と解釈し、仮のシフトカウントとする。つぎに x_2 の値がその n ビット符号つき (あるいは符号なし) 数として表現出来るなら仮のカウントを正式なシフトカウントとする。そうでない場合、その挙動は d に依存し、$d = \texttt{D}$ なら表現出来なくてもその解釈された値をシフトカウントとする。$d = \texttt{U}$ ならシフトカウントを $\perp$ とし、シフト結果全体を $\perp$ とする。

よっつあるシフト命令において、左シフト、右シフトの区別はシフトカウントが正の場合の向きによる。符号つきシフトか符号なしシフトかの区別はシフトされる x_1 の値 v_1 の解釈による。

$$
\begin{aligned}
\mathcal{E}[\![(\texttt{LSHS}\ t\ x_1\ x_2)]\!] &\triangleq \mathcal{E}[\![(\texttt{LSHS}\ t\ x_1\ x_2\ \&\ (\texttt{U 5 D}))]\!] \\
\mathcal{E}[\![(\texttt{LSHS}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{genericshift}\,w\,(\texttt{zb}\,v_1)\,(\texttt{shiftcount}\,v_2\,[\![m]\!])) \\
\texttt{genericshift} &: w{:}\,\texttt{Bitw} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\,w \\
\texttt{genericshift}\,w\,n\,c &\triangleq \texttt{bz}\,w\,(\texttt{floor}(n * 2^c)) \\
\texttt{shiftcount} &: \texttt{Bits} \to \texttt{ShiftModifier} \to \boldsymbol{Z}_\perp \\
\texttt{shiftcount}\,b\,[\![(s\,n\,d)]\!] &\triangleq \textbf{if}\ c_0 = c_1\ \textbf{then}\ c_0\ \textbf{else} \\
&\quad\ \textbf{case}\ [\![d]\!]\ \textbf{of}\ \texttt{D} \Rightarrow c_1\ \texttt{U} \Rightarrow \perp \\
&\quad\ \textbf{where} \\
&\qquad f \triangleq \textbf{case}\ [\![s]\!]\ \textbf{of}\ \texttt{S} \Rightarrow \texttt{zb}\ \texttt{U} \Rightarrow \texttt{nb} \\
&\qquad c_0 \triangleq f\,b \\
&\qquad c_1 \triangleq f(\texttt{bz}\,n\,c_0)
\end{aligned}
$$

22. $(\texttt{LSHU}\ t\ x_1\ x_2\ [\&\ m])$ $\quad t, t_2 \in \texttt{Itype}\ \wedge\ t = t_1$

符号なし左シフト。

$$
\begin{aligned}
\mathcal{E}[\![(\texttt{LSHU}\ t\ x_1\ x_2)]\!] &\triangleq \mathcal{E}[\![(\texttt{LSHU}\ t\ x_1\ x_2\ \&\ (\texttt{U 5 D}))]\!] \\
\mathcal{E}[\![(\texttt{LSHU}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{genericshift}\,w\,(\texttt{nb}\,v_1)\,(\texttt{shiftcount}\,v_2\,[\![m]\!]))
\end{aligned}
$$

23. $(\texttt{RSHS}\ t\ x_1\ x_2\ [\&\ m])$ $\quad t, t_2 \in \texttt{Itype}\ \wedge\ t = t_1$

符号つき右シフト。

$$
\begin{aligned}
\mathcal{E}[\![(\texttt{RSHS}\ t\ x_1\ x_2)]\!] &\triangleq \mathcal{E}[\![(\texttt{RSHS}\ t\ x_1\ x_2\ \&\ (\texttt{U 5 D}))]\!] \\
\mathcal{E}[\![(\texttt{RSHS}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{genericshift}\,w\,(\texttt{zb}\,v_1)\,(-\,\texttt{shiftcount}\,v_2\,[\![m]\!]))
\end{aligned}
$$

24. $(\texttt{RSHU}\ t\ x_1\ x_2\ [\&\ m])$ $\quad t, t_2 \in \texttt{Itype}\ \wedge\ t = t_1$

符号なし右シフト。

$$
\begin{aligned}
\mathcal{E}[\![(\texttt{RSHU}\ t\ x_1\ x_2)]\!] &\triangleq \mathcal{E}[\![(\texttt{RSHU}\ t\ x_1\ x_2\ \&\ (\texttt{U 5 D}))]\!] \\
\mathcal{E}[\![(\texttt{RSHU}\ t\ x_1\ x_2\ \&\ m)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{genericshift}\,w\,(\texttt{nb}\,v_1)\,(-\,\texttt{shiftcount}\,v_2\,[\![m]\!]))
\end{aligned}
$$

25. $(\texttt{TSTEQ}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype}\ \wedge\ t_1 = t_2$

比較 $(x_1 = x_2)$。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{TSTEQ}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x = y)\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Itype}\\
\mathcal{E}[\![(\texttt{TSTEQ}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x = y)\,(\texttt{rb}\,v_1)\,(\texttt{rb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Ftype}\\
\texttt{tstr} &: w{:}\texttt{Bitw} \to (\boldsymbol{R} \to \boldsymbol{R} \to \boldsymbol{Bool}) \to \boldsymbol{R} \to \boldsymbol{R} \to \texttt{Bit}\,w\\
\texttt{tstr}\,w\,f\,x\,y &\triangleq \texttt{bz}\,w\,\textbf{if } f\,x\,y\,\textbf{ then } 1\,\textbf{ else } 0
\end{aligned}$$

26. $(\texttt{TSTNE}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \wedge t_1 = t_2$

比較 $(x_1 \neq x_2)$。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{TSTNE}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x \neq y)\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Itype}\\
\mathcal{E}[\![(\texttt{TSTNE}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x \neq y)\,(\texttt{rb}\,v_1)\,(\texttt{rb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Ftype}
\end{aligned}$$

27. $(\texttt{TSTLTS}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \wedge t_1 = t_2$

符号つき比較 $(x_1 < x_2)$。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{TSTLTS}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x < y)\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Itype}\\
\mathcal{E}[\![(\texttt{TSTLTS}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x < y)\,(\texttt{rb}\,v_1)\,(\texttt{rb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Ftype}
\end{aligned}$$

28. $(\texttt{TSTLES}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \wedge t_1 = t_2$

符号つき比較 $(x_1 \leq x_2)$。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{TSTLES}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x \leq y)\,(\texttt{zb}\,v_1)\,(\texttt{zb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Itype}\\
\mathcal{E}[\![(\texttt{TSTLES}\ t\ x_1\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x \leq y)\,(\texttt{rb}\,v_1)\,(\texttt{rb}\,v_2)) \quad \textbf{if } t_1 \in \texttt{Ftype}
\end{aligned}$$

29. $(\texttt{TSTGTS}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \wedge t_1 = t_2$

符号つき比較 $(x_1 > x_2)$。

$$\mathcal{E}[\![(\texttt{TSTGTS}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLTS}\ t\ x_2\ x_1)]\!]$$

30. $(\texttt{TSTGES}\ t\ x_1\ x_2)$ $\quad t \in \texttt{Itype} \wedge t_1 = t_2$

符号つき比較 $(x_1 \geq x_2)$。

$$\mathcal{E}[\![(\texttt{TSTGES}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLES}\ t\ x_2\ x_1)]\!]$$

31. $(\texttt{TSTLTU}\ t\ x_1\ x_2)$ $\quad t, t_1 \in \texttt{Itype} \wedge t_1 = t_2$

符号なし比較 $(x_1 < x_2)$。

$$\mathcal{E}[\![(\texttt{TSTLTU}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x < y)\,(\texttt{nb}\,v_1)\,(\texttt{nb}\,v_2))$$

32. $(\texttt{TSTLEU}\ t\ x_1\ x_2)$ $\quad t, t_1 \in \texttt{Itype} \wedge t_1 = t_2$

符号なし比較 $(x_1 \leq x_2)$。

$$\mathcal{E}[\![(\texttt{TSTLEU}\ t\ x_1\ x_2)]\!]\rho\,\sigma \triangleq (\sigma_2, \texttt{tstr}\,w\,(\lambda xy.x \leq y)\,(\texttt{nb}\,v_1)\,(\texttt{nb}\,v_2))$$

33. $(\texttt{TSTGTU}\ t\ x_1\ x_2)$ $\quad t, t_1 \in \texttt{Itype}\ \wedge\ t_1 = t_2$

符号なし比較 $(x_1 > x_2)$。

$$\mathcal{E}[\![(\texttt{TSTGTU}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLTU}\ t\ x_2\ x_1)]\!]$$

34. $(\texttt{TSTGEU}\ t\ x_1\ x_2)$ $\quad t, t_1 \in \texttt{Itype}\ \wedge\ t_1 = t_2$

符号なし比較 $(x_1 \geq x_2)$。

$$\mathcal{E}[\![(\texttt{TSTGEU}\ t\ x_1\ x_2)]\!] \triangleq \mathcal{E}[\![(\texttt{TSTLEU}\ t\ x_2\ x_1)]\!]$$

35. $(\texttt{ASMCONST}\ t\ x_1)$ $\quad t = t_1\ \wedge\ \texttt{isasmconst}\ x_1$

意味は x_1 と同じである。

$$\begin{aligned}\mathcal{E}[\![(\texttt{ASMCONST}\ t\ x_1)]\!] &\triangleq \mathcal{E}[\![x_1]\!]\\ \texttt{isasmconst} &: \texttt{Lexp} \to \boldsymbol{Bool}\\ \texttt{isasmconst} &\triangleq \text{Unspecified}\end{aligned}$$

ここに isasmconst はリンカが解決出来る定数かどうかを判定する Unspecified 関数であり、以下の条件を満たさなければならない。

$$\forall e \in \texttt{Lexp}\quad \texttt{isasmconst}\, e\ \rightarrow\ \forall x \in^*_{\mathbf{a}} e\ \ x \in \texttt{ConstExp} + \texttt{StaticExp} + \texttt{PureExp}$$

- 議論

Q. この条件では ASMCONST 式がネストできることになるが。

A. 式変形中のことを考慮し、validexp では禁止していない。コード生成フェーズではいずれ別の制約条件が必要であり、そこでは ASMCONST 式のネストも禁止される。

36. $(\texttt{PURE}\ t\ x_1\ \cdots\ x_n)$ $\quad x_1 \in \texttt{IntConstExp}$

副作用のない任意の演算。拡張用。

$$\begin{aligned}\mathcal{E}[\![(\texttt{PURE}\ t\ x_1\ \cdots\ x_n)]\!]\rho\,\sigma &\triangleq (\sigma_n, \texttt{pureapply}[\![t]\!]\,(\texttt{nb}\, v_1)\,[v_2, \ldots, v_n])\\ \texttt{pureapply} &: t{:}\texttt{Ltype} \to \boldsymbol{N} \to [\texttt{Bits}] \to \mathcal{T}[\![t]\!]\\ \texttt{pureapply} &\triangleq \text{Unspecified}\end{aligned}$$

ここに pureapply は自然数 $\texttt{nb}\, v_1$ で示される関数に引数 $[v_2, \ldots, v_n]$ を与えた結果を計算する Unspecified 関数である。

5.6.5 Mem 式

ここでは Mem 式に対する $\mathcal{E}$ を定義する。

1. $(\texttt{MEM}\ t\ x_1\ [\&\ m])$ $\quad t_1 \in \texttt{Itype}$

データメモリのアドレス x_1 にある型 t のオブジェクトが格納している値。$m \in$ `MemModifier` が存在し、$m =$ `V` の時にかぎり Mem 式は volatile オブジェクトを表し、読み込みがトレースに記録され、結果は予測不能値となる。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{MEM}\ t\ x_1)]\!] &\triangleq \mathcal{E}[\![(\texttt{MEM}\ t\ x_1\ \&\ \texttt{N})]\!] \\
\mathcal{E}[\![(\texttt{MEM}\ t\ x_1\ \&\ m)]\!]\rho\,\sigma &\triangleq \textbf{case}\ [\![m]\!]\ \textbf{of} \\
&\quad \texttt{N} \Rightarrow (\sigma_1, \texttt{dmread}\,\sigma_1\,(\texttt{nb}\,v_1)\,[\![t]\!]) \\
&\quad \texttt{V} \Rightarrow \texttt{random}[\![t]\!]\,(\texttt{addtotr}\,\sigma_1\,\texttt{READ}\,[\![t]\!]\,[v_1])
\end{aligned}$$

5.6.6 Set 式

ここでは Set 式に対する $\mathcal{E}$ を定義する。

1. $(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ [\&\ m])\ x_2) \quad t = t' = t_2$

   データメモリのアドレス x_1 にある型 t のオブジェクトへ x_2 の値を代入する。$m \in$ `MemModifier` が存在し、$m =$ `V` の時にかぎり Mem 式は volatile オブジェクトを表し、書き込みがトレースに記録される。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1)\ x_2)]\!] &\triangleq \mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ \&\ \texttt{N})\ x_2)]\!] \\
\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ \&\ m)\ x_2)]\!]\rho\,\sigma &\triangleq (\sigma_4, v_2) \\
&\quad \textbf{where} \\
&\quad \sigma_3 \triangleq \texttt{dmwrite}\,\sigma_2(\texttt{nb}\,v_1)\,[\![t]\!]\,v_2 \\
&\quad \sigma_4 \triangleq \textbf{case}\ [\![m]\!]\ \textbf{of} \\
&\qquad \texttt{N} \Rightarrow \sigma_3 \\
&\qquad \texttt{V} \Rightarrow \texttt{addtotr}\,\sigma_3\,\texttt{WRITE}\,[\![t]\!]\,[v_1, v_2]
\end{aligned}$$

2. $(\texttt{SET}\ t\ (\texttt{REG}\ t'\ s)\ x_1) \qquad t = t' = t_1$

   Reg 式が表すレジスタに x_1 の値を代入する。

$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{REG}\ t'\ s)\ x_1)]\!]\rho\,\sigma \triangleq (\texttt{rmwrite}\,\sigma_1\,(\rho.\texttt{reg}[\![s]\!])\,[\![t]\!]\,v_1, v_1)$$

3. $(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n\ [\&\ m])\ x_1) \quad t = t' = t_1$

   SubReg 式が表す部分レジスタに x_1 の値を代入する。$m \in$ `SubregModifier` が存在し、$m =$ `N`

の時にかぎり (REG t_1 s) 中の指定部分以外に予測不能値が入る。

$$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n)\ x_1)]\!] \triangleq \mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n\ \&\ \texttt{S})\ x_1)]\!]$$

$$\begin{aligned}
\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{SUBREG}\ t'\ (\texttt{REG}\ t_1\ s)\ n\ \&\ m)\ x_1)]\!]\rho\,\sigma &\triangleq (\sigma_4, v_1) \\
&\quad \textbf{where} \\
&\quad\ a \triangleq \rho.\texttt{reg}[\![s]\!] + \texttt{subregoffset}[\![t]\!]\,[\![t_1]\!]\,n \\
&\quad\ \sigma_3 \triangleq \textbf{case}\ [\![m]\!]\ \textbf{of} \\
&\qquad\qquad \texttt{S} \Rightarrow \sigma_1 \\
&\qquad\qquad \texttt{N} \Rightarrow \texttt{rmwrite}\ \sigma_2\,(\rho.\texttt{reg}[\![s]\!])\,[\![t]\!]\,r \\
&\qquad\qquad\qquad \textbf{where}\ (\sigma_2, r) \triangleq \texttt{random}[\![t]\!]\,\sigma_1 \\
&\quad\ \sigma_4 \triangleq \texttt{rmwrite}\,\sigma_3\,a\,[\![t]\!]\,v_1
\end{aligned}$$

5.6.7 Jump 式

ここでは Jump 式に対する $\mathcal{E}$ を定義する。簡潔さのため、ここの意味記述において以下を定義する。

$$j_i \triangleq \mathcal{E}[\![l_i]\!]\rho\sigma$$

1. (JUMP l_1)

   ラベル式 l_1 で示されるロケーションへジャンプする。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{JUMP}\ l_1)]\!]\rho\,\sigma &\triangleq (\texttt{jmp}\,\sigma\,j_1, \emptyset) \\
\texttt{jmp} &: \texttt{Mem} \to \texttt{Bits} \to \texttt{Mem} \\
\texttt{jmp}\,\sigma\,j &\triangleq \sigma := \{.\texttt{pc} = \texttt{nb}\,j\}
\end{aligned}$$

2. (JUMPC x_1 l_1 l_2) $\qquad x_1.\texttt{code} \in \texttt{TstOps}$

   Tst 式 x_1 の値が 1 なら l_1 へ、0 なら l_2 へジャンプする。

$$\mathcal{E}[\![(\texttt{JUMPC}\ x_1\ l_1\ l_2)]\!]\rho\,\sigma \triangleq (\texttt{jmp}\,\sigma_1\ \textbf{case}\ \texttt{nb}\,v_1\ \textbf{of}\ 1 \Rightarrow j_1\ 0 \Rightarrow j_2, \emptyset)$$

   - 議論

   Q. x_1 が比較演算に限られている理由は。
   A. あることはあるが、この件はもう少し検討が必要である。

3. (JUMPN x_1 $((c_1\ l_1)\cdots(c_n\ l_n))$ l_0) $\quad t_1 \in \texttt{Itype} \wedge i \neq j \to c_i \neq c_j$

   整数式 x_1 の値が $c_i \in \boldsymbol{Z}$ に等しければ l_i へジャンプ、そうでなければ l_0 へジャンプする。

$$\mathcal{E}[\![(\texttt{JUMPN}\ x_1\ ((c_1\ l_1)\cdots(c_n\ l_n))\ l_0)]\!]\rho\,\sigma \triangleq (\texttt{jmp}\,\sigma_1\ \textbf{if}\ \exists!i\ c_i = \texttt{nb}\,v_1\ \textbf{then}\ j_i\ \textbf{else}\ j_0, \emptyset)$$

5.6.8 DefLabel 式

ここでは DefLabel 式に対する $\mathcal{E}$ を定義する。DefLabel 式は L 環境の LabelEnv を作るために mklabelenv で参照されるが、$\mathcal{E}$ では単に無視される。

1. (DEFLABEL s)

$$\mathcal{E}[\![(\texttt{DEFLABEL}\ s)]\!]\rho\,\sigma \triangleq (\sigma, \emptyset)$$

5.6.9 Call 式

ここでは Call 式に対する $\mathcal{E}$ を定義する。

1. (CALL x_1 (x_2 $\cdots$ x_n) (y_1 $\cdots$ y_m))　$t_1 \in$ Itype $\wedge$ ★ y_i は「相異なる」 InterfaceVar

   x_1 の値 v_1 で示されるプログラムメモリに L 関数 $\sigma.\mathtt{pm}\,v_1 (\neq \perp_{\mathtt{Lfunc}})$ があると仮定し、多値関数としての解釈 $\mathcal{F}[\![\sigma.\mathtt{pm}\,v_1]\!]$ に引数を与え、結果の多値を y_i に代入する。

$$\begin{aligned}
\mathcal{E}[\![(\texttt{CALL}\ x_1\ (x_2\ \cdots\ x_n)\ (y_1\ \cdots\ y_m))]\!]\rho\,\sigma \triangleq\ & (\sigma'_m, \emptyset) \\
& \textbf{where} \\
(\sigma'_0, [b_1, \ldots, b_m]) \triangleq\ & \mathcal{F}[\![\sigma.\mathtt{pm}\,v_1]\!]\,\rho\,[v_2, \ldots, v_n]\,\sigma_n \\
t_i \triangleq\ & y_i.\mathtt{type} \\
b'_i \triangleq\ & \mathtt{mkconst}[\![t_i]\!]\,b_i \\
(\sigma'_i, u_i) \triangleq\ & \mathcal{E}[\![(\texttt{SET}\ t_i\ y_i\ b'_i)]\!]\,\rho\,\sigma'_{i-1}
\end{aligned}$$

5.6.10 Interface 式

★まだ。

ここでは Interface 式に対する $\mathcal{E}$ を定義する。$\mathcal{E}$ で参照されるのは w_f のみである。他は $\mathcal{F}$ で参照される。

1. (PROLOGUE (w_f w_r) x_1 $\cdots$ x_n)　★ x_i は「相異なる」InterfaceVar

$$\mathcal{E}[\![(\texttt{PROLOGUE}\ x_1\ \cdots\ x_n)]\!]\rho\,\sigma \triangleq \text{★フレームポインタを } w_f \text{ だけインクリメント。}$$

2. (EPILOGUE (w_f w_r) x_1 $\cdots$ x_n)

$$\mathcal{E}[\![(\texttt{EPILOGUE}\ x_1\ \cdots\ x_n)]\!]\rho\,\sigma \triangleq \text{★フレームポインタを } w_f \text{ だけデクリメント。}$$

★ $\mathcal{F}$ のセマンティックスにおけるローカルなエリアの変更を無視することと関連しており、きちんとした定義はやっかいである。こまったもんだ。

5.6.11 Special 式

ここでは Special 式に対する $\mathcal{E}$ を定義する。以下の定義では副作用 x_i と x_j ($i < j$ とする) が競合している場合、すなわち同一のオブジェクトを変更しようとしている場合、後の副作用 x_j が有効であるということも表している。

1. (PARALLEL $x_1 \ \cdots \ x_n$)

   $x_1, \ldots, x_n$ の副作用を同時に行なう。

$$\begin{aligned}\mathcal{E}[\![(\texttt{PARALLEL}\ x_1\ \cdots\ x_n)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ & (\sigma := [\sigma_1 \div \sigma, \ldots, \sigma_n \div \sigma], \emptyset) \\ & \textbf{where} \\ & (\sigma_i, v_i) \ \stackrel{\triangle}{=}\ \mathcal{E}[\![x_i]\!]\rho\,\sigma\end{aligned}$$

2. (USE r)

   レジスタ r が使用されることを表す。

$$\mathcal{E}[\![(\texttt{USE}\ r)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ \bigstar\text{こまったもんだ}$$

3. (CLOBBER x_1)

   構文から x_1 は Set 式の第一引数に許されるものであり、そこに予測不能値が入ることを表す。

$$\begin{aligned}\mathcal{E}[\![(\texttt{CLOBBER}\ x_1)]\!]\rho\,\sigma \ \stackrel{\triangle}{=}\ & \mathcal{E}[\![(\texttt{SET}\ t_1\ x_1\ c)]\!]\rho\,\sigma_2 \\ & \textbf{where} \\ & (\sigma_2, r) \ \stackrel{\triangle}{=}\ \texttt{random}[\![t_1]\!]\,\sigma_1 \\ & c \ \stackrel{\triangle}{=}\ \texttt{mkconst}[\![t_1]\!]\,r\end{aligned}$$

5.7 L 連想リストのセマンティックス ($\mathcal{A}$)

L 連想リスト $a \in \texttt{Lalist}$ のセマンティックス $\mathcal{A}[\![a]\!]$ は引数のエントリクラス (c: EntClass) に依存した型 (EntType c) のレコードで、それは文字列からそのいろいろな属性を返す関数のレコードである。validalist の制約から L 連想リストのエントリは重複したエントリ名を持たないと仮定している。

$$
\begin{aligned}
\mathcal{A} &: \texttt{Lalist} \to c{:}\,\texttt{EntClass} \to \texttt{EntType}\,c \\
\texttt{EntType} &: \texttt{EntClass} \to \boldsymbol{Type} \\
\texttt{EntType REG} &\triangleq \{\ \texttt{type} : \texttt{String} \to \texttt{Ltype}_\perp, \\
&\qquad \texttt{offset} : \texttt{String} \to \texttt{Offset}_\perp\} \\
\texttt{EntType FRAME} &\triangleq \{\ \texttt{type} : \texttt{String} \to \texttt{Ltype}_\perp, \\
&\qquad \texttt{align} : \texttt{String} \to \texttt{Align}_\perp, \\
&\qquad \texttt{offset} : \texttt{String} \to \texttt{Offset}_\perp\} \\
\texttt{EntType REG} &\triangleq \{\ \texttt{type} : \texttt{String} \to (\texttt{Ltype} + \{\texttt{UNKNOWN}\})_\perp, \\
&\qquad \texttt{align} : \texttt{String} \to \texttt{Align}_\perp, \\
&\qquad \texttt{segment} : \texttt{String} \to \texttt{Segment}_\perp, \\
&\qquad \texttt{linkage} : \texttt{String} \to \texttt{Linkage}_\perp\ \}
\end{aligned}
$$

以下は L 連想リストから特定のエントリクラスのエントリのみをリストとして取り出すための補助関数 entfilter の「つまらない定義」である。

$$
\begin{aligned}
\texttt{entfilter} &: \texttt{EntClass} \to \texttt{Lalist} \to \texttt{Ent List} \\
\texttt{entfilter}\,c\,[\![(\texttt{ALIST})]\!] &\triangleq () \\
\texttt{entfilter}\,c\,[\![(\texttt{ALIST}\ e_0\ e_1\ \cdots)]\!] &\triangleq (\textbf{if}\ c = e_0.\texttt{class}\ \textbf{then}\ [\![(e_0)]\!]\ \textbf{else}\ [\![()]\!]) \\
&\qquad +\!+\ \texttt{entfilter}\,c\,[\![(\texttt{ALIST}\ e_1\ \cdots)]\!]
\end{aligned}
$$

以下が $\mathcal{A}$ の定義である。

$$
\begin{aligned}
\mathcal{A}[\![a]\!]\,\texttt{REG} &\triangleq \{\ \texttt{type} = \{n_i \mapsto t_i\}, \\
&\qquad \texttt{offset} = \{n_i \mapsto o_i\}\ \} \\
&\qquad \textbf{where}\quad ((n_1\ \texttt{REG}\ t_1\ o_1)\ \cdots) \triangleq \texttt{entfilter REG}\,[\![a]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}[\![a]\!]\,\texttt{FRAME} &\triangleq \{\ \texttt{type} = \{n_i \mapsto t_i\}, \\
&\qquad \texttt{align} = \{n_i \mapsto a_i\}, \\
&\qquad \texttt{offset} = \{n_i \mapsto o_i\}\ \} \\
&\qquad \textbf{where}\quad ((n_1\ \texttt{FRAME}\ t_1\ a_1\ o_1)\ \cdots) \triangleq \texttt{entfilter FRAME}\,[\![a]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}[\![a]\!]\,\texttt{STATIC} &\triangleq \{\ \texttt{type} = \{n_i \mapsto t_i\}, \\
&\qquad \texttt{align} = \{n_i \mapsto a_i\}, \\
&\qquad \texttt{segment} = \{n_i \mapsto s_i\}, \\
&\qquad \texttt{linkage} = \{n_i \mapsto l_i\}\ \} \\
&\qquad \textbf{where}\quad ((n_1\ \texttt{STATIC}\ t_1\ a_1\ s_1\ l_1)\ \cdots) \triangleq \texttt{entfilter STATIC}\,[\![a]\!]
\end{aligned}
$$

5.8 L 関数のセマンティックス ($\mathcal{F}$)

ここでは意味関数 $\mathcal{F}$ を定義する。ここに Args, Rets はそれぞれ引数と返却値の型である。L 関数は多値関数なので、返却値も Bits のリストになっている。

$$\begin{aligned}
\mathcal{F} &: \mathtt{Lfunc} \to \mathtt{Env} \to \mathtt{Args} \to \mathtt{Mem} \to (\mathtt{Mem} \times \mathtt{Rets})_\perp \\
\mathtt{Args}, \mathtt{Rets} &: \boldsymbol{Type} \\
\mathtt{Args} &\triangleq [\mathtt{Bits}] \\
\mathtt{Rets} &\triangleq [\mathtt{Bits}]
\end{aligned}$$

L 関数 $f \in \mathtt{Lfunc}$ の意味 $\mathcal{F}[\![f]\!]$ は環境、実引数、メモリを受けとり、修正されたメモリと多値関数としての返却値の組を返す関数である。L 関数内の L 連想リストでローカルに宣言されているフレーム変数とレジスタは関数の実行中のみ存在する。フレーム変数については Interface 式によるフレームポインタのメンテナンスにより、これが実現される。レジスタについては以下の Unspecified 関数により、Prologue 式の w_r バイトだけレジスタメモリに新たなエリアを確保する。

$$\begin{aligned}
\mathtt{newregframe} &: \mathtt{RMem} \to \boldsymbol{N} \to \boldsymbol{N} \\
\mathtt{newregframe} &\triangleq \text{Unspecified}
\end{aligned}$$

ここに newregframe はレジスタメモリ m:RMem とバイト数 n を取り、集合 $\mathtt{RMemAddr} - \mathtt{dom}\, r$ 中の連続した n バイトをある方法で探し、そのエリアのアドレスを返す。すなわち newregframe は以下の条件を満たさなければならない。

$$\forall x \ 0 \le x < n \ \to \mathtt{newregframe}\, m\ n\ + x \in \mathtt{RMemAddr} - \mathtt{dom}\, r$$

以下は $\mathcal{F}$ の定義である。f_newenv により関数内の環境を作り、Prologue 式を $\mathcal{E}$ で評価してフレームポインタを設定し、f_args により実引数を仮引数に代入し、f_exec により本体を実行し、f_rets により Epilogue 式にある返却値の L 式のならびを評価して多値を得、Epilogue 式を $\mathcal{E}$ で評価してフレームポインタを戻す。最後に f_newmem により、ローカルな変更 (フレームとレジスタメモリ) 以外の変更を反映させたメモリを作り、多値の返却値と組にして返す。

$$\begin{aligned}
&\mathcal{F}[\![(\mathtt{FUNCTION}\ name\ alist\ seq@(pro\ \cdots\ epi))]\!]\,\rho\,[a_1,\ldots,a_n]\,\sigma \triangleq (\sigma_6, [b_1,\ldots,b_m]) \\
&\textbf{where} \\
&(\mathtt{PROLOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n) \triangleq pro \\
&(\mathtt{EPILOGUE}\ (w_f\ w_r)\ y_1\ \cdots\ y_m) \triangleq epi \\
&\rho' \triangleq \mathtt{f_newenv}\,\rho\,\sigma.\mathtt{rm}\ alist\ seq\ w_r \\
&(\sigma_1, v_1) \triangleq \mathcal{E}[\![pro]\!]\,\rho\,\sigma \\
&\sigma_2 \triangleq \mathtt{f_args}[\![(x_1\ \cdots\ x_n)]\!]\,\rho'\,\sigma_1\,[a_1,\ldots,a_n] \\
&\sigma_3 \triangleq \mathtt{f_exec}\,seq\,\rho'\,(\sigma_2 := \{.\mathtt{pc} = 1\}) \\
&(\sigma_4, [b_1,\ldots,b_m]) \triangleq \mathtt{f_rets}[\![(y_1\ \cdots\ y_m)]\!]\,\rho'\,\sigma_3 \\
&(\sigma_5, v_2) \triangleq \mathcal{E}[\![epi]\!]\,\rho\,\sigma_4 \\
&\sigma_6 \triangleq \mathtt{f_newmem}\,\sigma_5\,\sigma
\end{aligned}$$

この定義中 **where** 以下の f_ で始まる関数は全てローカルであり、他からは参照されないが、組版の都合上それらを以下に分けて定義する。

- f_newenv は呼びだし側からの環境 ρ を一部修正し、L 関数でローカルな L 環境を作る。

$$
\begin{aligned}
\texttt{f\_newenv} &: \texttt{Env} \to \texttt{RMem} \to \texttt{Lalist} \to \texttt{Lseq} \to \boldsymbol{N} \to \texttt{Env} \\
\texttt{f\_newenv}\,\rho\ m\ a\ s\ w_r &\triangleq \rho := \{.\texttt{reg} = r, .\texttt{fra} = f, .\texttt{lab} = l\} \\
&\quad \textbf{where} \\
&\quad\ r : \texttt{RegEnv} \\
&\quad\ r\,s \triangleq \texttt{newregframe}\,m\ w_r\ +\ (\mathcal{A}[\![a]\!]\,\texttt{REG}).\texttt{offset}\ s \\
&\quad\ f : \texttt{FrameEnv} \\
&\quad\ f \triangleq \rho.\texttt{fra} := (\mathcal{A}[\![a]\!]\,\texttt{FRAME}).\texttt{offset} \\
&\quad\ l : \texttt{LabelEnv} \\
&\quad\ l \triangleq \texttt{mklabelenv}\,seq
\end{aligned}
$$

- f_args は引数の結合を行なう。

$$
\begin{aligned}
\texttt{f\_args} &: \texttt{InterfaceVar List} \to \texttt{Env} \to \texttt{Mem} \to \texttt{Args} \to \texttt{Mem}_\perp \\
\texttt{f\_args}[\![(x_1\ \cdots\ x_n)]\!]\,\rho\,\sigma_0\,[a_1,\ldots,a_n] &\triangleq \sigma_n \\
&\quad \textbf{where} \\
&\qquad t_i \triangleq x_i.\texttt{type} \\
&\qquad a'_i \triangleq \texttt{mkconst}[\![t_i]\!]\,a_i \\
&\qquad (\sigma_i, j_i) \triangleq \mathcal{E}[\![(\texttt{SET}\ t_i\ x_i\ a'_i)]\!]\,\rho\,\sigma_{i-1}
\end{aligned}
$$

- f_exec は Epilogue 式の直前まで実行。

$$
\begin{aligned}
\texttt{f\_exec} &: \texttt{Lseq} \to \texttt{Env} \to \texttt{Mem} \to \texttt{Mem}_\perp \\
\texttt{f\_exec}[\![s]\!]\,\rho\,\sigma &\triangleq \textbf{if}\ s\,!\,\sigma.\texttt{pc} \in \texttt{EpilogueExp}\ \textbf{then}\ \sigma\ \textbf{else}\ \texttt{f\_exec}[\![s]\!]\,\rho\,(\mathcal{S}[\![s]\!]\,\rho\,\sigma\,!\,0)
\end{aligned}
$$

- f_rets は Epilogue 式の多値を評価。

$$
\begin{aligned}
\texttt{f\_rets} &: \texttt{TypedExp List} \to \texttt{Env} \to \texttt{Mem} \to (\texttt{Mem} \times \texttt{Rets})_\perp \\
\texttt{f\_rets}[\![(y_1\ \cdots\ y_m)]\!]\,\rho\,\sigma_0 &\triangleq (\sigma_m, [b_1,\ldots,b_m]) \\
&\quad \textbf{where} \\
&\quad (\sigma_i, b_i) \triangleq \mathcal{E}[\![y_i]\!]\,\rho\,\sigma_{i-1}
\end{aligned}
$$

- f_newmem は関数呼びだし後の L メモリを、ローカルに変更されたデータメモリとレジスタメモリのドメインを元の σ のものに制限することにより作る。

$$
\begin{aligned}
\texttt{f\_newmem} &: \texttt{Mem} \to \texttt{Mem} \to \texttt{Mem} \\
\texttt{f\_newmem}\,\sigma'\,\sigma &\triangleq \sigma' := \{.\texttt{rm} = \sigma'.\texttt{rm}\,|_{\texttt{dom}\,\sigma.\texttt{rm}}, .\texttt{dm} = \sigma'.\texttt{dm}\,|_{\texttt{dom}\,\sigma.\texttt{dm}}\}
\end{aligned}
$$

★ $\mathcal{F}$ による L 関数の等価性の定義。ローカルは無視しているから一時変数やレジスタなどの違いは意味の違いにならない。

5.9 L プログラムのセマンティックス $(\mathcal{P}, \mathcal{M}, \mathcal{D})$

★まだ。

$\mathcal{P}$: Lprog $\rightarrow$ Env $\times$ Mem
$\mathcal{M}$: ★リロケータブルオブジェクトを抽象化したもの。
$\mathcal{D}$: ★リロケータブルデータを抽象化したもの。

6 volatile 変数のセマンティックスの妥当性について

マシンはメモリを介して外部とのやりとりを行なう．従って、そのポートとして使われる変数についての最適化は制限される．C 言語ではそのような変数、一般にはオブジェクトを volatile という一種の型により表現することが出来る．LIR では `& V` というモディファイアを伴った `MEM` 式により volatile オブジェクトを表す．このセクションでは volatile オブジェクトの意味定義を解説し、またその定義が自然なものであることを示す．

6.1 ランダムステートによる意味付け

ここで `v1,v2` がなんらかの装置とのポートとして使われている volatile 変数と仮定し、どのような変換が許されないかを考えてみる．まず明らかに以下のような変換は禁止しなければならない．これはパイプラインスケジューラなどがやるような最適化変換である．

```
t1 = v1        ×        v2 = 1
v2 = 1       ——→       t1 = v1
  Rule 1: 順序を変えてはならない
```

以下も同様に禁止しなければならない．これは ded code elimination で行なわれるような最適化変換である．

```
v1 = 1         ×
v1 = 2       ——→       v1 = 2
   Rule 2: 重複を省いてはならない
```

装置が volatile オブジェクトを読むことにより動作し、また外部からそこへの書き込まれる可能性もあるので以下の変換も禁止される．これは共通部分式と式の簡略化で行なわれるような最適化変換である．

```
t1 = v1        ×        junk = v1
t2 = v1      ——→        junk = v1
t3 = t1-t2                t3 = 0
   Rule 3: 値はいつも違う
```

よって、volatile オブジェクトの「意味」としては、以上のような変換がその意味を変えてしまうような定義をしなければならない．そのために、意味定義におけるメモリーには以下のようなふたつのフィールド、トレース及びランダムステートが存在し、トレースにより Rule 1,2 を、ランダムステートにより Rule 3 を反映させている．

$$\texttt{Mem} \triangleq \{\cdots, \texttt{tr}\colon \texttt{Trace}, \texttt{rs}\colon \boldsymbol{R}\}$$

ここにトレースはアクションのリストである．また新たなランダムステートを求めるための関数 `random` があるとする．

$$\begin{aligned} \texttt{Trace} &\triangleq [\texttt{Action}] \\ \texttt{Action} &\triangleq \texttt{Symbol} \times \texttt{Ltype} \times [\texttt{Bits}] \\ \texttt{random} &: t\colon \texttt{Ltype} \to \texttt{Mem} \to \texttt{Mem} \times \mathcal{T}[\![t]\!] \end{aligned}$$

処理	Action
読み込み	(READ, $ltype$, $[address]$)
書き込み	(WRITE, $ltype$, $[address, value]$)

そして volatile メモリーの参照ではそのアドレスがトレースに記録され、値はランダムステートとなる.

$$
\begin{array}{l}
\mathcal{E}[\![(\texttt{MEM}\, t\ x_1\ \texttt{\&}\ m)]\!] \\
\triangleq \\
\textbf{case}\ [\![m]\!]\ \textbf{of} \\
\quad \texttt{N} \Rightarrow (\sigma_1, \texttt{dmread}\,\sigma_1\,(\texttt{nb}\,v_1)\,[\![t]\!]) \\
\quad \texttt{V} \Rightarrow \texttt{random}[\![t]\!]\,(\texttt{addtotr}\,\sigma_1\,\texttt{READ}\,[\![t]\!]\,[v_1]) \\
\quad \textbf{where} \quad (\sigma_1, v_1) \stackrel{\triangle}{=} \mathcal{E}[\![x_1]\!]\rho\sigma
\end{array}
$$

また volatile メモリーへの書き込みではそのアドレスと値がトレースに記録される.

$$
\begin{array}{l}
\mathcal{E}[\![(\texttt{SET}\, t\ (\texttt{MEM}\, t'\ x_1\ \texttt{\&}\ m)\ x_2)]\!] \stackrel{\triangle}{=} (\sigma_4, v_2) \\
\textbf{where} \\
(\sigma_1, v_1) \stackrel{\triangle}{=} \mathcal{E}[\![x_1]\!]\rho\sigma \\
(\sigma_2, v_2) \stackrel{\triangle}{=} \mathcal{E}[\![x_2]\!]\rho\sigma_1 \\
\qquad \sigma_3 \stackrel{\triangle}{=} \texttt{dmwrite}\,\sigma_2(\texttt{nb}\,v_1)\,[\![t]\!]\,v_2 \\
\qquad \sigma_4 \stackrel{\triangle}{=} \textbf{case}\ [\![m]\!]\ \textbf{of} \\
\qquad\qquad \texttt{N} \Rightarrow \sigma_3 \\
\qquad\qquad \texttt{V} \Rightarrow \texttt{addtotr}\,\sigma_3\,\texttt{WRITE}[\![t]\!][v_1, v_2]
\end{array}
$$

6.2 意味定義の妥当性

このトレースとランダムステートに基づいた意味定義は Rule 1 から Rule 3 にのみ基づいたものであり、その妥当性に疑問がある．何か volatile 変数のより自然なモデルからの裏付がほしい．LIR の実行可能なプログラム（以下 L プログラム）は volatile 変数をチャネルとして動作するプロセスと見なせる．ここではこの視点で我々の意味定義の妥当性を議論する.

はじめに、bisimulation [17] に基づいた L プログラムの等価性を定義する．その定式化は coalgebra[19][18] で行なう．次に我々が与えた表示的意味、すなわちトレースとランダムステートに基づく volatile 変数の意味に従った L プログラムの第二の等価性を定義する．最初の等価性を coalgebra 等価性、第二の等価性をトレース等価性という．そして、このふたつの等価性は同等であることを証明する.

まず以下の議論で使用する概念を要約する．必要なら bisimulation については [17]、coalgebra については [19]、カテゴリー理論については [16][?] 等を参照されたい.

$F{:}\mathcal{C} \to \mathcal{C}$ をカテゴリー $\mathcal{C}$ 上のファンクターとするとき、F-coalgebra とはオブジェクト S と射 $\alpha{:}\, S \to F(S)$ の組 (S, α) である．このとき S をその coalgebra のキャリアと呼ぶ．ファンクター F が明らかな場合、$F-$ を略し単に coalgebra とも呼ぶ．以下を可換にする $f{:}\, S \to T$ を (S, α) から (T, β) への F-homomorphism という.

$$
\begin{array}{ccc}
S & \xrightarrow{\ f\ } & T \\
{\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\
F(S) & \xrightarrow[F(f)]{} & F(T)
\end{array}
$$

与えられたカテゴリー $\mathcal{C}$ とファンクター F について、F-coalgebra をオブジェクト、F-homomorphism を射とするカテゴリーを $\mathcal{C}_F$ で表す．このとき、$\mathcal{C}$ を $\mathcal{C}_F$ の underlying category という．$\mathcal{C}_F$ において

以下の (R,γ) を (S,α) と (T,β) の F-bisimulation という．またこのような (R,γ) が存在するとき、(S,α) と (T,β) は bisimilar であるという．

$$(S,\alpha) \longleftarrow (R,\gamma) \longrightarrow (T,\beta)$$

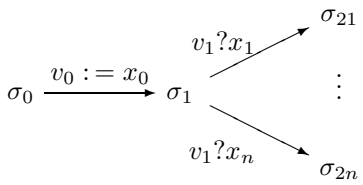
以下の議論で underlying category は $\mathcal{S}et$ である．この場合、上の条件はは以下の図式の可換性と同一である．ここに $R \subset S \times T$、$\pi_1 : (s,t) \mapsto s$、$\pi_2 : (s,t) \mapsto t$.

$$\begin{array}{ccccc} S & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & T \\ \downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\gamma} & & \downarrow{\scriptstyle\beta} \\ F(S) & \xleftarrow[F(\pi_1)]{} & F(R) & \xrightarrow[F(\pi_2)]{} & F(T) \end{array}$$

6.3 coalgebra 等価性

coalgebra 等価性は L プログラムのプロセスを以下に述べる方法でラベル付き遷移系と見なした時の bisimulation 等価性である．

我々の意味定義におけるメモリ $\sigma \in \texttt{Mem}$ はプログラムメモリとデータメモリを含んでいるので、σ をプログラムとも呼ぶ．以下の遷移図における $\sigma_i \in \texttt{Mem}$ は全て同一のプログラムであり、遷移によってデータメモリのみが変化している．



ラベル $v := x$ は volatile 変数 v へ値 x を書き込むことによる遷移を意味する． ラベル $v?x_i$ は変数 v の値が x_i である場合の遷移を意味する．いずれの遷移も決定的である．volatile 変数は外部から書き込みが起こり得るが、それは $v?x_i$ による遷移の「直前」に起こると考える．また、以後の議論では各 LIR 命令（以下 L 式）は高々一つの volatile アクセス（読み書き）を持つと仮定する．LIR では部分式の評価順序が定められているので、これらの仮定は本質的な制限にはならない．

L プログラムを以上のようなラベル付き遷移系と見なすと、bisimulation によりその自然な等価性が定義される．我々はこの等価性をフォーマライズの容易さのために coalgebra により表現する．上記のラベル付き遷移系を coalgebra で表すために、ファンクター F、キャリア X、射 $\alpha : X \to F(X)$ を以下のように定義する．

$$\begin{array}{rcl} F & : & \text{Functor on the categeory } \mathcal{S}et \\ F(X) & \triangleq & \texttt{Vol} \times \texttt{Bits} \times X \\ & & +\texttt{Vol} \times \mathrm{Hom}(\texttt{Bits}, X) \\ & & +\{\bot\} \\ \texttt{Vol} & \triangleq & \text{Set of volatile addresses} \\ \texttt{Memc} & \triangleq & \texttt{Mem} \text{ without } \texttt{.tr} \text{ and } \texttt{.rs} \\ \mathcal{S}c & : & \texttt{Memc} \to \texttt{Memc} \\ \alpha & : & \texttt{Memc} \to F(\texttt{Memc}) \end{array}$$

ここに $\mathtt{Vol} \subset \boldsymbol{N}$ は volatile オブジェクトのアドレスの集合である．ファンクター F は三つのファンクターの直和であり、最初のものが $v := x$、第二が $v?x$ に、そして $\bot$ は volatile オブジェクトにアクセスしないまま動き続けるプロセスを意味する．coalgebra 等価性ではトレースとランダムステートは参照しないので、それらを Mem から除いたものがキャリア Memc である．Sc は LIR の仕様（付録）で定義されている意味関数 S をトレースとランダムステートを無視するように修正したものである．以下の議論では環境 Env は本質的ではないので、それも除外している．α はこの Sc により volatile アクセス単位で以下のような実行をする．まず volatile アクセスの直前まで実行を進め、そのアクセスが $x \in \mathtt{Bits}$ を $v \in \mathtt{Vol}$ で示されるオブジェクトへ書き込む命令ならば α は (v, x, σ') を返す．ここに σ' は書き込み後のメモリである．またもしその volatile アクセスが $v \in \mathtt{Vol}$ からの読み込みの場合は $(v, f{:}\mathtt{Bits} \to \mathtt{Memc})$ を返す．ここに $f\, x_i$ は先のラベル付き遷移系において、ラベル $v?x_i$ による遷移先の σ である．

以上で我々は LIR におけるプログラム全体の coalgebra を定義したことになる．特定のプログラム σ の coalgebra はその部分 coalgebra $(\overline{\sigma}, \alpha)$ であり、そのキャリア $\overline{\sigma}$ は以下のように定義される．

$$
\begin{aligned}
\overline{\sigma} &\triangleq \{\sigma' \in \mathtt{Memc} \mid \sigma \xrightarrow{\alpha^*} \sigma'\} \\
\sigma \xrightarrow{\alpha^*} \sigma' &\triangleq \sigma = \sigma' \vee \\
&\quad \exists \sigma''\ \sigma \xrightarrow{\alpha^*} \sigma'' \wedge \sigma'' \xrightarrow{\alpha} \sigma' \\
\sigma \xrightarrow{\alpha} \sigma' &\triangleq \alpha\,\sigma \in \mathtt{Vol} \times \mathtt{Bits} \times \mathtt{Memc} \\
&\qquad \wedge \alpha\,\sigma\,!\,2 = \sigma' \\
&\quad \vee \\
&\quad \alpha\,\sigma \in \mathtt{Vol} \times \mathrm{Hom}(\mathtt{Bits}, \mathtt{Memc}) \\
&\qquad \wedge \exists b\ (\alpha\,\sigma\,!\,1)b = \sigma'
\end{aligned}
$$

以上より coalgebra 等価性を次のように定義する．

> プログラム $\sigma_1, \sigma_2 \in \mathtt{Memc}$ から作られる部分 coalgebra $(\overline{\sigma_1}, \alpha)$ と $(\overline{\sigma_2}, \alpha)$ が bisimilar のとき、σ_1 と σ_2 は coalgebra 等価であるといい、$\sigma_1 \stackrel{c}{=} \sigma_2$ と表す．

この等価性の定義は OS のように停止しないで動き続けるプログラムの等価性も含んだ自然なものである．一方我々が与えた LIR の表示的意味では停止しないプログラムは全て $\bot$ という扱いになってしまう．しかしプログラムが停止する場合には、これらの二つの等価性は等価であることが示される．

6.4 トレース等価性

はじめに LIR の本来の意味論すなわちトレースとランダムステートに基づいたプログラムの等価性（トレース等価性）を定義する必要がある．そのために、プログラムはある特定のメイン関数から開始するものとする．まず以下の定義をおこなう．

$$
\begin{aligned}
\beta &: \mathtt{Mem} \to \mathtt{Mem}_\bot \\
\beta\,\tau &\triangleq \text{execute until the next volatile} \\
\mathtt{t} &: \mathtt{Mem} \to \boldsymbol{R} \to \mathtt{Trace} \\
\mathtt{t}\,\tau\,r &\triangleq [a_1, \ldots, a_n] \\
\mathtt{c} &: \mathtt{Mem} \to \mathtt{Memc} \\
\mathtt{c}\,\tau &\triangleq \tau \text{ without random states and traces}
\end{aligned}
$$

ここに β は α と同様の volatile アクセス単位での実行関数であるが、これは本来のトレースとランダムステートを使った意味関数 $\mathcal{E}$ と $\mathcal{S}$ に基づくものである．関数 t はプログラム τ の全体の挙動として

のトレースを返す．引数 r は τ の実行におけるランダムステートの初期値である．関数 $\mathtt{c}$ はプログラム τ からランダムステートとトレースを無視（削除）するものである．

プログラム τ の挙動は乱数関数 random 及びランダムステートの初期値 $r \in \boldsymbol{R}$ で完全に決定される．しかし random の実現の仕方が高々加算であることから、ある特別な random 関数を fix し、可能なあらゆるランダム列を初期値の違いでのみ表現することは可能である．よって以後プログラムの挙動は単に初期値 r で完全に決定されるとして議論する．以下がトレース等価性の定義である．

> プログラム $\tau_1, \tau_2 \in \mathtt{Mem}$ が以下を満たす時、
>
> $$\forall r \in \boldsymbol{R}\ \mathtt{t}\,\tau_1\, r = \mathtt{t}\,\tau_2\, r$$
>
> それらはトレース等価であるといい、 $\tau_1 \overset{t}{=} \tau_2$ と表す．

6.5 二つの等価性の同値性

以下の定理は coalgebra 等価性とトレース等価性が同値であるということを意味する．証明はスケッチである．

定理 ($\overset{t}{=}$ と $\overset{c}{=}$ は同値)

$$\forall \tau_1, \tau_2 \in \mathtt{Mem}\ \ \tau_1 \overset{t}{=} \tau_2 \leftrightarrow \mathtt{c}\,\tau_1 \overset{c}{=} \mathtt{c}\,\tau_2$$

証明

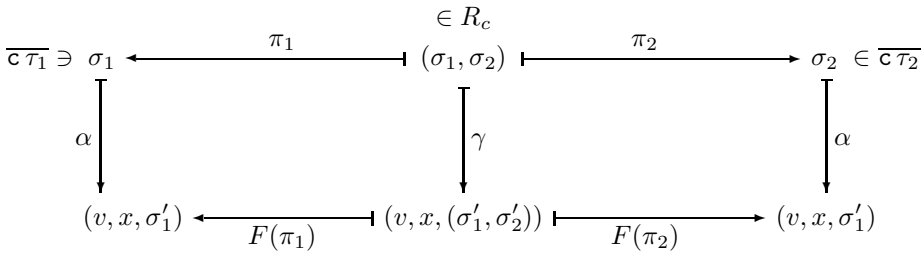
プログラム $\tau_1, \tau_2 \in \mathtt{Mem}$ が $\tau_1 \overset{t}{=} \tau_2$ を満たすと仮定し、以下を定義する．

$$\begin{aligned} R_t &\overset{\triangle}{=} \{(\tau_1', \tau_2') \in \mathtt{Mem}^2 \mid \\ &\qquad \exists r \in \boldsymbol{R} \\ &\qquad (\tau_i := \{.\mathtt{rs} = r\}) \xrightarrow{\beta^*} \tau_i'\ (i = 1, 2)\} \\ R_c &\overset{\triangle}{=} \{(\mathtt{c}\,\tau_1, \mathtt{c}\,\tau_2) \in \mathtt{Memc}^2 \mid (\tau_1, \tau_2) \in R_t\} \end{aligned}$$

遷移関数の定義をフォーマルには与えていないが、 $(\tau_1, \tau_2) \in R_t$ ならば $\beta{:}\,\tau_1 \to \tau_1'$ と $\beta{:}\,\tau_2 \to \tau_2'$ が同じアクション、すなわち供に書き込みか読み込みか $\perp$ であることは明らかである．

以下 R_c が $(\overline{\mathtt{c}\,\tau_1}, \alpha)$ と $(\overline{\mathtt{c}\,\tau_2}, \alpha)$ のある bisimulation であることを F の場合分けで示す．

1. $(\mathtt{Vol} \times \mathtt{Bits} \times (-))$ の場合:



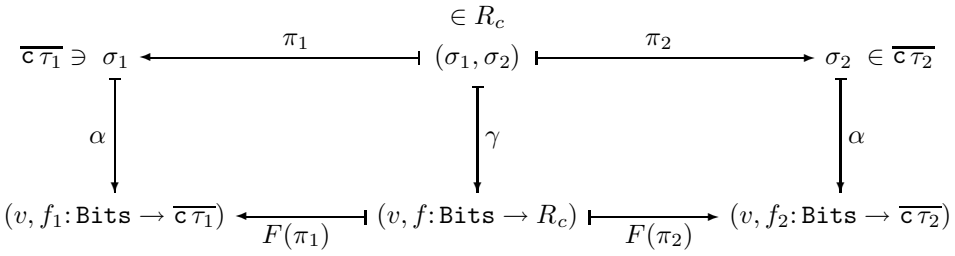
R_c の定義より、ある $\tau_1', \tau_2' \in R_t$ が存在して $\sigma_i = \mathtt{c}\,\tau_i'\ \ (i = 1, 2)$ となる．今の場合、τ_i の選択は任意である．

R_t の定義より $\tau_1', \tau_2' \in R_t$ のアクションは等しい．例えばそれが $v := x$ であれば、以下が成り立つ．

$$\alpha\colon \mathtt{c}\,\tau_i' \mapsto (v, x, \mathtt{c}\,\tau_i'') \quad (i = 1, 2)$$

すなわち図 1 の下の左右は正しい．よってもし $(\sigma_1', \sigma_2') \in R_c$ であれば、γ を図式どうりに定義できるが、上の議論より $(\sigma_1', \sigma_2') = (\mathtt{c}\,\tau_1'', \mathtt{c}\,\tau_2'')$ であり R_t の定義から $(\tau_1'', \tau_2'') \in R_t$ よって $(\sigma_1', \sigma_2') \in R_c$ であり、最初の場合は示された．

2. $(\mathtt{Vol} \times \mathrm{Hom}(\mathtt{Bits}, -))$ の場合:



同様の議論により、この図式の可換性はほぼ明らかである．証明を完結するには γ の定義が必要である．もし任意の $x \in \mathtt{Bits}$ について $(f_1\,x, f_2\,x) \in R_c$ であれば、$f \stackrel{\triangle}{=} f_1 \times f_2$ として、γ が定義出来る．そのような $x \in \mathtt{Bits}$ を固定し、$\sigma_i = \mathtt{c}\,\tau_i'$ でかつ $\beta\colon \tau_i' \mapsto \tau_i''$ のアクションが $v?x$ となるように $\tau_1', \tau_2' \in R_t$ を選ぶ．これは R_t の定義及び乱数の初期値のみで全ての挙動が決定するという議論から可能である．よって $f_i\,x = \mathtt{c}\,\tau_i''$ $(i = 1, 2)$ すなわち $(\tau_1'', \tau_2'') \in R_t$ である． 逆の証明、及び $\perp$ の場合については省略する．

7 LIR への追加要求

1. 整数型 I1,I2,I3
2. SUBREG で MEM 式も許す
3. 普通の条件ジャンプ (リストラ後でコードの順番も決まった後を想定)
4. (JUMPC x1 x2) x1 が 0 以外ならジャンプ (x2 は LABEL 式または REG 式または MEM 式)
5. 遅延分岐など

   (DELAYEDJUMP x) 上と同様だが次の命令の実行後にジャンプ

   (DELAYEDJUMPC x1 x2) 上と同様だが次の命令の実行後にジャンプ

   (ANNULEDJUMPC x1 x2) 上と同様だがジャンプしないときは次の命令をスキップ
6. 変換

   (CONVFID t x) 浮動小数点数を整数へ (設定したモード)
   (CONVFIDD t x) 浮動小数点数を整数へ (設定したモードでオーバーフローしたら 80...h)
   (CONVFIR t x) 浮動小数点数を整数へ (直近値)
   (CONVFIN t x) 浮動小数点数を整数へ (切り捨て)
   (CONVFIP t x) 浮動小数点数を整数へ (切り上げ)
   (CONVFIZ t x) 浮動小数点数を整数へ (ゼロ方向)
   (CONVSF t x) 符号つき整数を浮動小数点数へ
   (CONVUF t x) 符号なし整数を浮動小数点数へ
   (ROUNDD t x) 浮動小数点数のまま整数化 (設定したモード)
   (ROUNDR t x) 浮動小数点数のまま整数化 (直近値)
   (ROUNDN t x) 浮動小数点数のまま整数化 (切り捨て)
   (ROUNDP t x) 浮動小数点数のまま整数化 (切り上げ)
   (ROUNDZ t x) 浮動小数点数のまま整数化 (ゼロ方向)

7. フラグ計算用演算子 (x1 と x2 は同じ型、x3 は 1 ビットでキャリーフラグを想定)

   (CARRY I1 x1 x2 x3)
   (OVERFLOW I1 x1 x2 x3)
   (ACARRY I1 x1 x2 x3) x86 の補助キャリー
   (PARITY I1 x) x のパリティが奇数なら 0
   (NOTANUMBER t x) x が非数か (x は浮動小数点型)
   (FSIGN I1 x) x の符号ビット (x は浮動小数点型で、0、無限大、非数でも可)

8. 条件判定

   (IFTHENELSE t x1 x2 x3) x1 の型は I1、x2,x3 の型は t で、x1 が 1 なら x2、0 なら x3

参考文献

[1] Andrew W.Appel,*Modern Compiler Implementation in ML*, Cambridge Univ. Press, 1998.

[2] Andrew W.Appel,*Compiling with Continuations*, Cambridge Univ. Press, 1992.

[3] Richard M.Stallman,*Using and Porting GNU CC*, Free Software Foundation, 1999.

[4] Steven S.Muchnick,*Advanced Compiler Design and Implemantation*, Morgan Kaufman, 199x.

[5] 中田育男,「コンパイラの構成と最適化」, 朝倉書店,1999.

[6] Guy L.Steele Jr.,*Common Lisp The Language, Second Edition*, Digital Press, 1990.

[7] B. Kernighan, D. Ritchie,*C Programming Language*, Prentice Hall, 1988.

[8] Allen Leung, Lal George: *Static Single Assignment Form for Machine* PLDI 99.

[9] Simon P.Jones, et al.: `C--`: A Portable Assembly Language, *Implementing Functional Languages 1997*, LNCS, 1998.

[10] Carl A.Gunter,*Semantics of Programming Languages*, MIT Press, 1992.

[11] L. Allison,*A Practical Introduction to Denotational Semantics*, Cambridge Univ. Press, 1986.

[12] R.Milner, et al. *The Definition of Standard ML*, The MIT Press, 1990.

[13] W.Clinger, J.Rees (eds),*Revised*^{4} *Report on The Algorithmic Language Scheme*, 1991.

[14] J.M.Spivey, *Understanding Z*, Cambridge Univ. Press, 1988.

[15] J.E.Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, The MIT Press, 1977.

[16] B.C.Pierce, *Basic Category Theory for Computer Scientists*, MIT Press, 1991.

[17] R.Milner, *A Calculus of Communicating Systems*, LNCS 92, 1980.

[18] P.Aczel and N.Mendler, A final coalgebra theorem, In *Proceedings Category Theory and Computer Science*, LNCS 389:357–365, 1989.

[19] J.J.M.M.Rutten, Universal coalgebra: a theory of systems, *Theoretical Computer Science*, 249:3–80,2000.

[20] E.P.de Vink, J.J.M.M.Rutten, Bisimulation for Probabilistic Transition Systems: a coalgebraic approach. *Theoretical Computer Science*, 221:3–80, 1999

索 引